Benjamin Wack
Sylvain Conchon, Judicaël Courant, Marc de Falco, Gilles Dowek,
Jean-Christophe Filliâtre et Stéphane Gonnord

Informatique
pour tous
en classes préparatoires aux grandes écoles

Manuel d'algorithmique et programmation structurée avec Python

Nouveaux programmes 2013

Voies MP, PC, PSI, PT, TPC et TSI

EYROLLES

Benjamin Wack

Sylvain Conchon, Judicaël Courant, Marc de Falco, Gilles Dowek,
Jean-Christophe Filliâtre et Stéphane Gonnord

Informatique pour tous en classes préparatoires aux grandes écoles

Manuel d'algorithmique et programmation structurée avec Python

Nouveaux programmes 2013
Voies MP, PC, PSI, PT, TPC et TSI

EYROLLES

Table des matières

Table des matières V

Première partie
Architecture matérielle et logicielle **1**

Chapitre 1
Machine, système d'exploitation et environnement de développement **3**

1.1 Qu'est-ce qu'un ordinateur ? 4
1.1.1 Observations externes 4
1.1.2 L'ordinateur, une machine universelle 4
1.1.3 Architecture des ordinateurs 7
1.2 Notion de système d'exploitation 13
1.2.1 Le multitâche 14
1.2.2 Identification des utilisateurs 14
1.2.3 Système de fichiers 16
1.2.4 Contrôle d'accès 19
1.2.5 Lancement d'applications 23
1.2.6 Protections 25
1.3 Environnement de développement intégré 25
1.3.1 Console interactive 27
1.3.2 Éditeur 29
1.3.3 Débogueur 30

Chapitre 2
Représentation des nombres **33**

2.1 Représentation des entiers naturels 34
2.1.1 Représentation de l'information par des booléens 34

2.1.2 La numération à position et les bases 34
2.1.3 La base deux 37
2.2 Représentation des entiers relatifs 39
2.2.1 Notation en complément à deux 39
2.2.2 Dépassements de capacité 41
2.3 Représentation des nombres à virgule 45
2.3.1 L'arithmétique flottante 45
2.3.2 Quelques cas particuliers 46
2.3.3 Dépassements de capacité et problèmes de précision 47
2.3.4 Les arrondis 48

DEUXIÈME PARTIE
Algorithmique et programmation **53**

CHAPITRE 3
Expressions : types et opérations **55**
3.1 Expressions et types simples 56
3.1.1 Expression 56
3.1.2 Entiers 57
3.1.3 Flottants 60
3.1.4 Booléens 64
3.2 Variables 68
3.2.1 Notion de variable 68
3.2.2 État et valeur d'une expression 68
3.2.3 Déclaration et initialisation 70
3.2.4 Affectation 71
3.3 Types composés 74
3.3.1 Les n-uplets 74
3.3.2 Chaînes de caractères : *strings* 77
3.3.3 Listes : une première approche 79
3.3.4 Conversions 80

CHAPITRE 4
Instructions : langage minimal de l'algorithmique **83**
4.1 Instructions 84
4.1.1 Notion d'algorithme 84
4.1.2 Notion de programme 84
4.1.3 Langage minimal de l'algorithmique 85
4.1.4 Entrées/sorties 85
4.1.5 Séquence d'instructions 86
4.2 Instructions conditionnelles 88
4.2.1 Test simple 88

4.2.2 Indentation signifiante . . . 88
4.2.3 Test avec alternative . . . 90
4.2.4 Tests imbriqués . . . 92
4.3 Boucles conditionnelles . . . 96
4.3.1 Nécessité des boucles . . . 96
4.3.2 Syntaxe d'une boucle conditionnelle . . . 96
4.3.3 Terminaison de boucle . . . 98
4.3.4 Invariant de boucle . . . 101
4.3.5 Boucle infinie . . . 102
4.4 Boucles inconditionnelles . . . 104
4.4.1 Boucle `for` . . . 104
4.4.2 Valeurs itérables . . . 106
4.4.3 L'itérable `range` . . . 107
4.4.4 Interrompre une boucle . . . 108
4.4.5 Boucles imbriquées . . . 109
4.5 Exercices . . . 111

CHAPITRE 5
Fonctions . . . **113**
5.1 La notion de fonction . . . 115
5.1.1 Le retour de valeur . . . 115
5.1.2 Variables globales et locales . . . 119
5.1.3 Ordre d'évaluation . . . 121
5.1.4 Passage par valeur . . . 122
5.2 Mécanismes avancés . . . 124
5.2.1 Fonctions locales . . . 124
5.2.2 Fonctions comme valeurs de première classe . . . 124
5.2.3 Fonctions partielles . . . 126
5.2.4 Fonctions de bibliothèque . . . 127
5.2.5 Méthodes . . . 129
5.3 La récursivité . . . 130
5.3.1 Concevoir une fonction récursive . . . 132
5.3.2 Terminaison et correction d'une fonction récursive . . . 135
5.3.3 Complexité d'une fonction récursive . . . 138
5.4 Exercices . . . 139

CHAPITRE 6
Notions de complexité et algorithmique sur les tableaux . . . **143**
6.1 Complexité d'un algorithme . . . 144
6.1.1 Plusieurs algorithmes pour un même problème . . . 144
6.1.2 Complexité et notation O . . . 146
6.1.3 Différentes nuances de complexité . . . 148

6.2 Structure de tableau . . . 150
6.2.1 Construction d'un tableau . . . 150
6.2.2 Accès aux éléments d'un tableau . . . 151
6.2.3 Parcours de tous les éléments d'un tableau . . . 153
6.3 Recherche dans un tableau . . . 155
6.3.1 Recherche séquentielle . . . 155
6.3.2 Recherche dichotomique dans un tableau trié . . . 156
6.4 Recherche d'un mot dans un texte . . . 158
6.5 Matrices . . . 160
6.5.1 Création . . . 162
6.5.2 Copie . . . 163
6.5.3 Dimensions . . . 164
6.5.4 Transposition . . . 164
6.5.5 Produit matriciel . . . 165
6.6 Mode de passage des tableaux . . . 166
6.7 Exercices . . . 168

TROISIÈME PARTIE
Ingénierie numérique et simulation . . . **171**

CHAPITRE 7
Pivot de Gauss et résolution de systèmes . . . **173**
7.1 Résolution de AX = Y : principe du pivot . . . 175
7.1.1 Le cas des systèmes triangulaires . . . 175
7.1.2 Les transvections . . . 176
7.1.3 Le problème de la comparaison à zéro . . . 177
7.1.4 Formalisation de l'algorithme . . . 179
7.1.5 Le formalisme matriciel . . . 180
7.2 Mise en œuvre . . . 182
7.2.1 Découper le travail . . . 182
7.2.2 Recoller les morceaux . . . 183
7.2.3 Comparaison avec numpy . . . 185
7.3 Complexité . . . 187
7.3.1 Mise sous forme triangulaire . . . 187
7.3.2 Phase de remontée . . . 188
7.3.3 Peut-on faire mieux que n^3 ? . . . 188
7.4 Conditionnement d'une matrice . . . 191
7.4.1 Mesurer les propagations d'erreurs . . . 191
7.4.2 Le conditionnement . . . 191
7.4.3 Quelques exemples caractéristiques . . . 192
7.5 Exercices . . . 194

CHAPITRE 8
Résolution numérique d'équations sur les réels **199**
8.1 Méthode dichotomique 200
8.1.1 Principe théorique 200
8.1.2 Terminaison, correction et complexité de l'algorithme 202
8.1.3 Mise en place, essais 203
8.2 Méthode de Newton 204
8.2.1 Extraction de racine 204
8.2.2 Algorithme général, terminaison, correction et complexité 206
8.2.3 Évaluation de la dérivée 207
8.2.4 Mise en œuvre 209
8.3 Quelle méthode choisir ? 210
8.3.1 Bien cerner le contexte 210
8.3.2 Utiliser `numpy/scipy` 211
8.4 Exercices 212

CHAPITRE 9
Résolution numérique d'équations différentielles **219**
9.1 Méthode d'Euler 220
9.1.1 Principe de la méthode d'Euler 220
9.1.2 Quelques notions d'analyse numérique 222
9.1.3 Choix du pas 224
9.2 Mise en œuvre 226
9.2.1 Équations scalaires d'ordre 1 226
9.2.2 Équations scalaires d'ordre 2 ou plus 228
9.3 Utilisation des bibliothèques `scipy` et `matplotlib` 230
9.3.1 Intégration des équations différentielles avec `odeint` 231
9.3.2 Représentation de graphes avec `plot` 232
9.3.3 De jolis graphes 234
9.3.4 Où on observe quelques limitations 237
9.4 Exercices 239

QUATRIÈME PARTIE
Bases de données **253**

CHAPITRE 10
Algèbre relationnelle **255**
10.1 Limites des structures de données plates pour la recherche d'informations 256
10.2 Représentation dans le modèle relationnel 257
10.3 Opérateurs sur le modèle relationnel 260
10.3.1 Description des recherches 260
10.3.2 Opérateurs ensemblistes usuels 261

10.3.3 Projection . . . 263
10.3.4 Sélection . . . 264
10.3.5 Renommage . . . 266
10.3.6 Algèbre relationnelle . . . 266
10.4 Utilisation d'un gestionnaire de bases de données relationnelles . . . 268
10.4.1 Description du langage SQL . . . 269
10.4.2 Projection . . . 269
10.4.3 Sélection . . . 269
10.4.4 Opérations ensemblistes . . . 270
10.4.5 Renommage . . . 270
10.5 Base de données et architecture logicielle . . . 271
10.5.1 Architecture client-serveur . . . 271
10.5.2 Architecture trois-tiers . . . 272
10.6 Exercices . . . 273

CHAPITRE 11
Base de données relationnelle . . . **275**
11.1 Clé primaire . . . 276
11.1.1 Clé . . . 276
11.1.2 Clé primaire . . . 276
11.1.3 Lien entre deux tables . . . 277
11.2 Opérateurs complexes de l'algèbre relationnelle . . . 278
11.2.1 Produit cartésien et division cartésienne . . . 279
11.2.2 Jointure . . . 281
11.2.3 Agrégation . . . 285
11.2.4 Composition de requêtes complexes . . . 289
11.3 Traduction en langage SQL . . . 290
11.3.1 Jointure . . . 290
11.3.2 Application simple d'une fonction d'agrégation . . . 291
11.3.3 Agrégation . . . 292
11.4 Exercices . . . 293

CINQUIÈME PARTIE
Algorithmique et programmation avancées . . . **297**

CHAPITRE 12
Structure de pile . . . **299**
12.1 Opérations caractérisant une structure de pile . . . 300
12.2 Réalisation d'une structure de pile . . . 302
12.2.1 Piles à capacité finie . . . 302
12.2.2 Piles non bornées . . . 304

12.3 Applications . . . 306
12.3.1 Analyse des mots bien parenthésés . . . 306
12.3.2 Évaluation d'une expression arithmétique en notation polonaise inverse . . . 308
12.3.3 Construction d'un labyrinthe parfait . . . 310
12.4 Exercices . . . 314

CHAPITRE 13
Algorithmes de tri . . . **315**
13.1 Tri par insertion . . . 316
13.1.1 Réalisation . . . 316
13.1.2 Complexité . . . 317
13.2 Tri rapide . . . 318
13.2.1 Réalisation . . . 318
13.2.2 Complexité . . . 321
13.3 Tri fusion . . . 322
13.3.1 Réalisation . . . 323
13.3.2 Complexité . . . 325
13.4 Exercices . . . 327

ANNEXE A
Travaux pratiques . . . **331**
A.1 Création de programmes autonomes . . . 331
A.1.1 Compilation d'un programme . . . 331
A.1.2 Exécution autonome d'un programme Python . . . 333
A.2 Mémoire virtuelle et performances de l'ordinateur . . . 334
A.3 Démontage d'un PC de bureau . . . 337
A.3.1 Sécurité . . . 337
A.3.2 Repérage des composants . . . 338
A.3.3 Mise en œuvre . . . 344
A.4 Résolution d'une équation du second degré avec gestion de la comparaison à zéro . . . 352
A.5 Représentation des nombres dans les calculatrices scientifiques . . . 352
A.6 Arithmétique et cryptographie . . . 354
A.6.1 Algorithme d'Euclide . . . 354
A.6.2 Décomposition en facteurs premiers . . . 355
A.6.3 Recherche de grands nombres premiers . . . 356
A.6.4 Application à la cryptographie : la méthode RSA . . . 358
A.7 Manipulation d'images bitmap . . . 359
A.7.1 Traitement pixel par pixel . . . 360
A.7.2 Traitement local . . . 361
A.7.3 Traitement global . . . 362
A.7.4 En couleurs . . . 362

A.8 Prise en main de phpMyAdmin 363
A.8.1 Création d'une table 363
A.8.2 Insertion de valeurs 366
A.9 Clés primaires et clés étrangères 368
A.9.1 Définition d'une clé primaire 368
A.9.2 Clé primaire auto-incrémentée 371
A.9.3 Lien entre deux tables 372
A.9.4 Lancement de requêtes 373

ANNEXE B
Compléments sur les entrées/sorties **375**
B.1 Lecture et écriture dans des fichiers 375
B.1.1 Lire les lignes d'un fichier 375
B.1.2 Extraction des données dans une ligne 376
B.1.3 Écrire des données dans un fichier 377
B.2 Lecture et écriture dans des images 379
B.2.1 Lecture d'image 379
B.2.2 Traitement 379
B.2.3 Écriture dans une image 380
B.3 Utilisation du module graphique turtle 380

Références **383**

Avant-propos

Proposer un enseignement spécifique d'informatique à tous les élèves de classes préparatoires aux grandes écoles scientifiques était une nécessité.

L'informatique est omniprésente dans le monde actuel. Chacun en a sa représentation personnelle, enthousiaste ou méfiante, superficielle ou pointue. Pour comprendre en profondeur ce qu'on entend par *informatique*, il faut commencer par clarifier ce qu'elle n'est pas.

Les anglophones la nomment souvent *computer science*, ce qui est un double contresens. Premièrement, parce que l'ordinateur n'est pas qu'une machine à calculer (*to compute* en anglais), en tout cas pas au sens où on l'entend dans le langage courant. Certes, les premières machines comme l'ENIAC ou l'EDVAC étaient utilisées exclusivement pour le calcul de tables balistiques. Cependant, il n'y a qu'à regarder fonctionner quelques minutes un ordinateur de bureau pour voir la diversité des tâches qu'il peut réaliser. Même si elle n'était pas bien exploitée par manque de ressources, cette polyvalence était déjà présente dans les tout premiers ordinateurs. Comme leur nom français l'indique, ce sont plutôt des machines à *ordonner* l'information, capables de stocker, de manipuler et de transmettre efficacement n'importe quelles données pourvu qu'elles leur soient fournies dans un format adéquat. Plutôt que de parler de polyvalence, on parlera donc d'*universalité* : un ordinateur peut traiter les données de toutes les façons raisonnables que l'on peut imaginer.

Ensuite, parce que l'informatique n'est pas uniquement la science des ordinateurs : elle est avant tout la science de l'*information* et de son traitement automatique. Elle démontre que tout objet du monde réel, et de certains mondes abstraits comme les mathématiques, peut se traduire par une représentation numérique, certes souvent imparfaite, mais que la recherche ne cesse d'améliorer au fil des années. La question du codage de l'information est bien antérieure à l'invention des ordinateurs, du dénombrement des moutons à l'aide de petits cailloux jusqu'au code morse, en passant par les différents systèmes d'écriture et de numération ou la programmation de motifs complexes dans les métiers à tisser.

Les ordinateurs n'ont fait que confirmer *a posteriori* la validité et l'utilité de ces représentations. Grâce à leur extraordinaire puissance de calcul, ils ont également donné en moins de cent ans un essor inédit à des modes de raisonnement patiemment élaborés pendant plusieurs millénaires. Cette pensée *algorithmique*, qui consiste à établir une méthode systématique pour résoudre un problème, était déjà connue des Mésopotamiens. Elle a trouvé une application directe lorsqu'il a fallu exprimer des façons de réaliser des processus complexes pour des machines très rapides mais dénuées d'initiative et de compréhension.

À peine née, la science informatique s'est confrontée à l'un de ses plus grands défis, qui l'occupe encore aujourd'hui : mettre au point des *langages* de communication communs à l'homme et à la machine. Comme les langues naturelles, ceux-ci doivent permettre une compréhension mutuelle entre deux entités ayant chacune leur propre représentation du monde, mais à un degré bien plus important puisqu'aucune place ne peut être laissée à l'ambiguïté.

L'informatique est donc bien plus qu'une technologie de pointe, c'est la science qui réunit ces quatre concepts de machine, d'information, d'algorithme et de langage et qui, les faisant travailler ensemble, leur a donné la place qu'ils occupent aujourd'hui dans nos sociétés.

L'étudiant qui se destine à une carrière d'ingénieur, d'enseignant ou de chercheur ne peut se dispenser d'une formation dans ce domaine tant il est incontournable dans presque toute activité professionnelle, en particulier dans les activités à caractère scientifique. En outre, cette formation ne peut se réduire à l'utilisation technique de logiciels : son contenu serait à la fois peu pérenne à cause de l'évolution rapide des outils, et d'une utilité très incertaine en fonction du parcours professionnel que l'on poursuivra.

Bien apprendre l'informatique demande surtout d'en saisir les concepts sous-jacents, qui restent valables malgré l'évolution des technologies. Pour ne donner que quelques exemples, l'architecture globale des ordinateurs n'a pas changé depuis plus de 60 ans ; les premiers ordinateurs représentaient déjà l'information en binaire ; les algorithmes de calcul numérique présentés dans cet ouvrage sont tous connus depuis le XVIII[e] siècle au moins ; les langages de programmation reposent presque tous sur les mêmes cinq instructions fondamentales. Il importe donc d'acquérir une culture informatique solide plutôt qu'un vernis technologique ; cela ne dispense pas de mettre très régulièrement en pratique les notions que l'on découvre, l'informatique ne pouvant s'apprendre qu'accompagnée d'une expérience régulière de programmation.

Enfin, même en dehors de tout cadre professionnel, le jeune citoyen qu'est l'élève de classe préparatoire sera fréquemment amené à rencontrer l'informatique dans sa vie quotidienne que ce soit par le biais des réseaux de communication, de ses loisirs, de ses achats, de ses interactions avec les administrations, etc. Comprendre comment fonctionnent ces systèmes et, à plus forte raison, en avoir soi-même programmé, même à un niveau modeste, est une clé indispensable pour en profiter en tant qu'acteur et pas seulement en tant que consommateur.

Structure de l'ouvrage

Le contenu de ce manuel se veut fidèle au programme officiel des deux années de classes préparatoires aux grandes écoles scientifiques. Il aborde les différentes notions qui sont pertinentes dans une formation scientifique, toujours avec la préoccupation de les replacer dans le contexte plus général de la science informatique. L'ensemble du contenu a vocation à être réutilisé pour le développement des Travaux d'Initiative Personnelle Encadrés.

- Dans la première partie **Architecture matérielle et logicielle**, on aborde les mécanismes internes d'un ordinateur. On présente les modèles théoriques qui régissent son fonctionnement, le système d'exploitation qui en permet l'usage quotidien, et les grands principes d'un environnement de programmation (**chapitre 1**). On donne ensuite un premier aperçu de la traduction numérique de l'information via la représentation des nombres en machine (**chapitre 2**), qui aura des conséquences importantes lorsqu'on voudra effectuer du calcul numérique.
- Dans la deuxième partie **Algorithmique et programmation**, on présente les notions clés de l'algorithmique (**chapitres 3 et 4**) en s'attachant systématiquement à démontrer que les algorithmes que l'on écrit produisent le résultat attendu. On aborde également la traduction de ces algorithmes sous forme de programmes. On présente ensuite la notion de fonction (**chapitre 5**) qui permet d'organiser les programmes et leur développement. Ce chapitre présente également les fonctions récursives, qui font partie du programme de deuxième année. On montre enfin comment évaluer l'efficacité d'un algorithme, et on présente une première structure de données : les tableaux (**chapitre 6**).
- Dans la troisième partie **Ingénierie numérique et simulation**, on étudie la traduction dans un langage de programmation d'algorithmes numériques abordés en cours de mathématiques : le pivot de Gauss pour la résolution de systèmes linéaires (**chapitre 7**), les méthodes de dichotomie et de Newton pour la résolution d'équations sur les réels (**chapitre 8**) et la méthode d'Euler pour la résolution d'équations différentielles (**chapitre 9**). Ces méthodes numériques mettent en lumière les limitations introduites par le passage sur machine. On présente enfin une utilisation raisonnée de bibliothèques de calcul.
- Dans la quatrième partie **Bases de données**, on s'intéresse à une représentation de l'information à la fois plus complexe et plus en lien avec les applications industrielles, par le biais du modèle relationnel des bases de données. On montre comment exprimer, dans le langage de l'algèbre relationnelle, des requêtes de recherche d'abord simples (**chapitre 10**), puis faisant intervenir plusieurs relations (**chapitre 11**) et on aborde la traduction de ces requêtes dans le langage SQL.
- La cinquième partie **Algorithmique et programmation avancées** couvre, avec la section sur les fonctions récursives du chapitre 5, le programme de deuxième année. On y montre qu'il existe d'autres structures de données comme la pile (**chapitre 12**) et on y compare plusieurs algorithmes de tri (**chapitre 13**) du point de vue de leurs complexités.

- On conclut ce manuel par une série de neuf propositions de travaux pratiques (**annexe A**) et par une brève documentation pratique sur quelques fonctions utiles au traitement de fichiers et à la production d'images (**annexe B**).

Chaque chapitre contient trois types de contenus :

- une partie de cours ;
- des sections intitulées « Savoir-faire », qui permettent d'acquérir les capacités essentielles ;
- des exercices, avec leur corrigé lorsque nécessaire. Les exercices les plus difficiles sont marqués d'un ou deux astérisques (*).

Trois types d'encadrés jalonnent cet ouvrage : **Attention** signale un piège ou une erreur fréquente chez les programmeurs débutants ; **En pratique** mentionne des considérations d'ordre pragmatique ; **Pour aller plus loin** propose des ouvertures vers des questions hors-programme.

Des compléments numériques à cet ouvrage sont proposés sur le site compagnon : `http://informatique-en-prepas.fr`.

Avertissement

Lorsqu'on conçoit un enseignement d'informatique, la question du choix du langage dans lequel on va programmer est incontournable, bien qu'*in fine* ce choix n'ait pas d'importance et que les compétences acquises dans un langage soient pour la plupart facilement transposables à un autre.

Le programme officiel de cet enseignement donne d'emblée la réponse à cette question puisqu'il impose le langage Python. Or ce langage existe en plusieurs versions incompatibles entre elles (un programme écrit pour une version ne fonctionnera pas toujours dans l'autre). On distingue notamment les versions 2.x (celles dont le numéro commence par 2) des versions 3.x, puisque c'est à la version 3.0 qu'ont eu lieu des changements incompatibles. Dans cet ouvrage, tous les programmes sont écrits en Python 3.x ; lorsque cela est nécessaire, un encadré précise les changements à apporter pour faire fonctionner ces programmes en Python 2.x.

Remerciements

Les auteurs tiennent à remercier Serge Abiteboul, Luc Albert, Jean-Pierre Archambault, Bruno Arsac, Jean-Philippe Berne, Hugues Bersini, Christophe Boilley, Sylvain Delpech, Francis Dorra, Pascal Lafourcade, Guillaume Le Blanc, Vincent Massart, Jean-Marie Monier, Pierre-Étienne Moreau, Franz Ridde, Landry Salle, Marie-Dominique Siefert et Thierry Viéville pour leur aide précieuse au cours de la rédaction de ce livre ainsi que l'équipe des éditions Eyrolles, Anne Bougnoux, Laurène Gibaud, Sébastien Mengin et Muriel Shan Sei Fan.

Première partie

Architecture matérielle et logicielle

Dans cette partie, nous abordons les mécanismes internes d'un ordinateur. Nous présentons les modèles théoriques qui régissent le fonctionnement d'un ordinateur, le système d'exploitation qui en permet l'usage quotidien, et les grands principes d'un environnement de programmation (chapitre 1). Nous donnons ensuite un premier aperçu de la traduction numérique de l'information via la représentation des nombres en machine (chapitre 2), qui aura des conséquences importantes lorsqu'on voudra effectuer du calcul numérique.

1

Machine, système d'exploitation et environnement de développement

Dans ce premier chapitre, nous préciserons ce qu'on peut désigner sous le terme d'ordinateur ou de machine numérique. Nous verrons que le fonctionnement d'un ordinateur est organisé au travers d'un système d'exploitation et qu'un environnement de développement permet d'y ajouter ses propres programmes.

1.1 Qu'est-ce qu'un ordinateur ?

Contrairement aux autres machines du quotidien, rien dans l'observation d'un ordinateur ne renseigne sur la façon dont il fonctionne. Comment savoir si telle machine est plus intéressante qu'une autre sans en comprendre le fonctionnement ?

Pour tout informaticien, une tablette numérique et un téléphone portable intelligent (smartphone) sont des ordinateurs, au même titre qu'un ordinateur de bureau. Pourquoi ?

Pour répondre à cette question, on observera dans un premier temps ces objets de l'extérieur. On verra que ces observations sont insuffisantes pour *caractériser* vraiment ce qu'est un ordinateur. En s'intéressant alors aux considérations qui motivent l'existence des ordinateurs, on expliquera comment les informaticiens peuvent apporter une réponse plus satisfaisante à cette question et on montrera comment cela conduit à une architecture commune à tous les ordinateurs actuels.

1.1.1 Observations externes

Considérons donc une tablette numérique, un smartphone et un ordinateur de bureau dans ce qu'ils ont de commun :

- Pour fonctionner, ils ont besoin d'une source d'énergie, en l'occurrence l'électricité.
- Ils reçoivent des informations de la part de l'utilisateur, par l'intermédiaire d'un clavier, d'une souris, d'un microphone, d'un écran tactile ou d'un réseau (téléphonique, wifi ou filaire).
- Ils émettent des informations, par l'intermédiaire de l'écran, de leur haut-parleur ou du réseau.

Or d'autres objets partagent ces caractéristiques. Ainsi une automobile a également besoin d'une source d'énergie pour fonctionner, reçoit des informations de son conducteur (accélérer, freiner, clignoter) et lui renvoie des informations (vitesse, niveau d'huile, température du moteur). Mais à la différence d'un ordinateur, elle a pour objectif principal de mouvoir ses utilisateurs et pas de traiter des informations.

Que penser alors d'un thermostat d'ambiance ? Ce dispositif fonctionne généralement sur piles, reçoit des informations (température de consigne, calendrier), a pour but principal de traiter ces dernières et transmet à la chaudière l'ordre de s'allumer ou de s'éteindre. Pourtant, on ne peut le considérer comme un ordinateur : il est trop spécialisé.

Pour qu'on puisse parler d'ordinateur, on voit que les caractéristiques évoquées, notamment la capacité de traiter des informations, sont nécessaires mais pas suffisantes.

1.1.2 L'ordinateur, une machine universelle

Il y a environ 5 000 ans, en Mésopotamie, les premières villes sont apparues. Il a fallu les organiser et, pour cela, gérer les informations les concernant : état des stocks, codification

des règles les régissant. C'est pour cela qu'ont été développés les systèmes de numération et l'écriture.

Plus récemment, on a commencé à réaliser que cette information pouvait être *traitée* de façon automatisée. Le métier à tisser mis au point par Jacquard en 1801 peut ainsi être programmé pour tisser des motifs complexes à l'aide de cartes perforées.

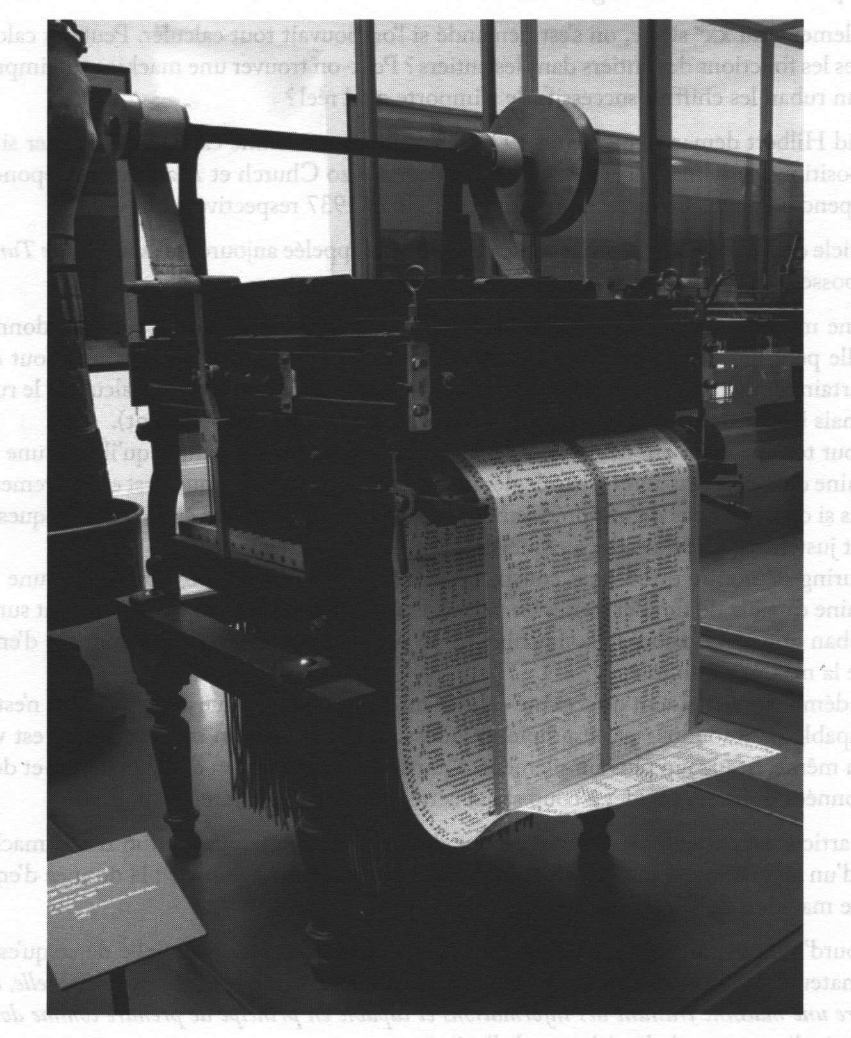


Figure 1.1
Mécanisme Jacquard au musée des arts et métiers de Paris

Les mathématiciens ont également réalisé que de nombreux calculs pourraient être automatisés. Dès 1642, Blaise Pascal avait ainsi conçu et réalisé une machine (la Pascaline) qui effectuait les quatre opérations usuelles sur les entiers : addition, soustraction, multiplication et division. Plus tard, vers le milieu du XIXᵉ siècle, suite aux travaux de Babbage, plusieurs *machines à différences*, des machines mécaniques, ont été produites pour calculer et imprimer des tables de logarithmes.

Finalement, au XXᵉ siècle, on s'est demandé si l'on pouvait tout calculer. Peut-on calculer toutes les fonctions des entiers dans les entiers ? Peut-on trouver une machine qui imprime sur un ruban les chiffres successifs de n'importe quel réel ?

David Hilbert demande même en 1928 s'il existe une machine capable de décider si une proposition mathématique est vraie ou fausse. Alonzo Church et Alan Turing répondent indépendamment non à cette question, en 1936 et 1937 respectivement.

L'article de Turing propose un modèle de machine (appelée aujourd'hui *machine de Turing*) qui possède les caractéristiques suivantes :

1 Une machine de Turing possède un ruban infini sur lequel on a disposé des données. Elle peut lire des données sur ce ruban, les traiter et en écrire d'autres. Au bout d'un certain temps, il se peut qu'elle s'arrête, on peut alors lire le résultat du calcul sur le ruban (mais il se peut aussi que la machine continue à travailler indéfiniment).
2 Pour tout procédé qui peut être calculé par un algorithme, il semble qu'il y ait une machine de Turing capable de le calculer (il est difficile de montrer que c'est effectivement le cas si on ne sait pas définir précisément ce qu'est un algorithme ; cette dernière question est justement une de celles auxquelles Turing tente de répondre).
3 Turing démontre qu'on peut construire une machine universelle, c'est-à-dire une machine capable de simuler toutes les autres. Pour l'utiliser, on dispose simplement sur son ruban une description de la machine qu'on veut simuler ainsi que les données d'entrée de la machine à simuler.
4 Il démontre également qu'il existe cependant des problèmes que cette machine n'est pas capable de résoudre : par exemple, décider si une proposition mathématique est vraie ou même, beaucoup plus simplement, à partir de la description d'une machine et de ses données d'entrée, décider à coup sûr si cette machine va s'arrêter ou non.

Cet article est extrêmement novateur car il considère que la description d'une machine (ou d'un algorithme) peut en fait être considérée comme une donnée : la donnée d'entrée d'une machine de Turing universelle.

Aujourd'hui, on considère ce point comme une caractérisation essentielle de ce qu'est un ordinateur : *un ordinateur est la réalisation concrète d'une machine de Turing universelle, c'est-à-dire une machine traitant des informations et capable en principe de prendre comme donnée d'entrée n'importe quel algorithme et de l'exécuter.*

Un ordinateur de bureau, une tablette et un smartphone sont bien des ordinateurs : on peut en effet leur faire exécuter des programmes arbitraires. Reste à expliquer comment ils calculent.

POUR ALLER PLUS LOIN Les systèmes embarqués

Un thermostat d'ambiance est-il une machine universelle ? S'il s'agit d'un thermostat mécanique, certainement pas. Et dans le cas d'un thermostat électronique ? Du point de vue de son utilisateur, ce n'est pas un ordinateur. Pourtant, il contient en son sein un véritable petit ordinateur, qu'on appelle généralement un microcontrôleur. Cependant, il est prévu pour exécuter le programme spécialisé qu'on lui a incorporé à la fabrication et non pour exécuter des algorithmes arbitraires. On peut donc le considérer comme un ordinateur manquant précisément de ce qui fait de lui un ordinateur !

De la même façon, un lecteur DVD de salon contient un ordinateur mais n'est pas fait non plus pour exécuter des programmes arbitraires. Quant au boîtier d'un fournisseur d'accès à Internet, c'est un ordinateur, dont le programme est régulièrement mis à jour. Cependant, l'utilisateur n'a pas le pouvoir de décider quel programme il exécutera.

On parle de systèmes embarqués pour désigner ces ordinateurs et programmes spécialisés.

1.1.3 Architecture des ordinateurs

Pour pouvoir exécuter des algorithmes arbitraires, les ordinateurs actuels sont tous bâtis autour du même modèle architectural théorique, l'architecture de von Neumann, même si la mise en œuvre pratique est un peu plus complexe.

L'architecture de von Neumann

Le premier ordinateur électronique conçu pour être une machine de Turing fut l'ENIAC (qui calculait des tables indiquant les paramètres de tir d'une batterie d'artillerie en fonction de la distance à la cible, du vent, etc.) dont la construction démarra en 1943. Son architecture a été décrite dans un rapport de John von Neumann en 1945 et est depuis appelée « architecture de von Neumann ». Depuis près de 70 ans, à quelques variations près, c'est l'architecture utilisée dans tous les ordinateurs, qu'il s'agisse de tablettes, smartphones, ordinateurs portables ou de bureau.

Une machine suivant l'architecture de von Neumann est constituée :

- d'une mémoire vive ;
- d'un processeur qu'on peut conceptuellement décomposer en une unité de contrôle et une unité de calcul arithmétique et logique (UAL) ;
- de dispositifs périphériques, appelés simplement périphériques ;
- d'un canal de communication entre la mémoire, le processeur et les périphériques, appelé le bus.

La mémoire vive est une suite de chiffres binaires (bits), organisés en pratique en octets (paquets de 8 bits) et en mots mémoire (64 bits sur les machines de bureau récentes). Elle a les caractéristiques essentielles suivantes :

- **Pas de sens a priori.** Un mot dans la mémoire peut aussi bien représenter une instruction d'un programme, qu'un nombre entier ou la couleur d'un point d'une image. La signification d'une valeur stockée dans la mémoire dépend donc entièrement de l'interprétation qu'en fait son utilisateur, ce dernier étant en l'occurrence autant le reste de l'ordinateur que l'être humain qui le commande.
- **Inertie.** La mémoire vive est inerte au sens où elle n'effectue aucun calcul.
- **Accès direct.** Tout mot mémoire possède une adresse (un nombre entier). On peut lire ou écrire directement le contenu d'un mot mémoire d'adresse donnée, d'où le nom donné en anglais à cette mémoire : *Random Access Memory (RAM).*

Les périphériques se présentent au reste de l'ordinateur sous la forme d'une mémoire supplémentaire : il s'agit de plages d'adresses sur lesquelles on peut écrire pour donner des ordres au périphérique, ou lire pour en obtenir des informations. À la différence de la mémoire vive, ils ne sont cependant pas nécessairement inertes et peuvent réagir aux instructions données : ainsi, une imprimante recevra des instructions via cette plage de mémoire, travaillera en conséquence et mettra certaines informations à disposition du reste de l'ordinateur (par exemple le niveau d'encre restant).

Le processeur est le cœur de l'ordinateur. De même qu'un chirurgien dans un bloc opératoire dirige les infirmiers pour qu'ils effectuent les tâches les plus simples pendant qu'il effectue le travail le plus délicat, le processeur donne des ordres aux périphériques et à la mémoire et est responsable de l'exécution du programme de l'ordinateur.

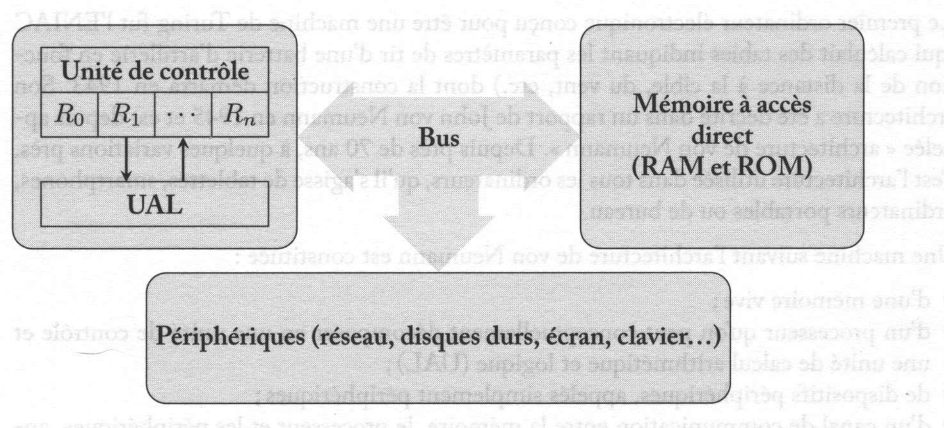


Figure 1.2
Architecture de von Neumann

Le processeur dispose d'une toute petite mémoire, typiquement de l'ordre de quelques mots à une centaine de mots, qu'on appelle des registres.

Grâce à son unité arithmétique et logique, il peut calculer la somme, la différence, le produit ou le rapport de nombres codés dans deux de ses registres et le stocker dans un troisième. Il peut également effectuer des opérations logiques (disjonction, conjonction, négation) sur des valeurs de vérité codées dans des registres.

Il peut enfin accéder à la mémoire, via le bus : pour tout couple de registres (R_1, R_2), il peut aller chercher dans la mémoire vive le contenu de la case mémoire dont l'adresse est stockée dans le registre R_1 et stocker le résultat dans R_2 ou inversement stocker le contenu de R_2 dans la case mémoire dont l'adresse est contenue dans R_1.

C'est l'unité de contrôle du processeur qui effectue ces actions. Cependant, cette unité ne contient pas le programme à exécuter : les instructions sont codées sous la forme d'une suite de bits stockée en mémoire. Plus précisément, l'unité de contrôle du processeur dispose d'un registre particulier, généralement appelé PC (pour *Program Counter*) et exécute en boucle la séquence d'actions suivante, résumée figure 1.3 :

1 **Lire instruction.** Aller lire le mot stocké à l'adresse mémoire donnée par le registre PC et le stocker dans un registre spécial appelé IR (*instruction register* ou registre d'instruction).
2 **Incrémenter PC.** Incrémenter le contenu du registre PC (c'est-à-dire ajouter 1 à la valeur et la stocker de nouveau dans le registre PC).
3 **Décoder instruction.** Décoder l'instruction contenue dans IR, c'est-à-dire interpréter la suite de bits contenue dans IR en une instruction que le processeur sait exécuter : opération arithmétique et logique, accès à la mémoire vive ou branchement (voir ci-dessous).
4 **Exécuter instruction.** Exécuter l'instruction décodée : opération arithmétique et logique, accès à la mémoire vive ou branchement.

On peut remarquer que si le registre PC contient initialement un entier n, son contenu après la séquence d'actions vaut $n + 1$. Lorsque le processeur effectue de nouveau cette séquence, il lit une nouvelle instruction en mémoire vive. Il exécute donc séquentiellement les instructions contenues dans des mots mémoires successifs.

Il existe cependant une exception d'importance : certaines instructions, appelées *instructions de branchement* ou *instructions de saut* peuvent modifier le registre PC. Elles stipulent que si une condition est réalisée (par exemple tel registre contient zéro/une valeur non nulle/une valeur positive/une valeur négative), alors le registre PC doit prendre une certaine valeur. Dans ce cas, lorsqu'il effectue de nouveau la séquence, le processeur continue donc son exécution en un autre point de la mémoire. Ces instructions de branchement sont utilisées d'une part pour réaliser des actions de façon conditionnelle (« *si telle condition est réalisée, alors effectuer telle séquence d'actions stockée à telle adresse* ») et, d'autre part,

pour effectuer des actions de façon répétitive (« *si tel registre contient une valeur non nulle, reprendre l'exécution à l'instruction stockée à telle adresse* »).

POUR ALLER PLUS LOIN Le pipeline

Cette présentation du processeur est en fait idéalisée.

D'une part, si sur les processeurs actuels un accès mémoire va bien chercher un mot, en revanche les adresses mémoire sont comptées en octets et non en mots. Lors de chaque cycle de décodage d'instructions, le PC est donc incrémenté de 4 sur un processeur 32 bits (un mot de 32 bits étant sur 4 octets) et de 8 sur un processeur 64 bits.

D'autre part, le processeur n'attend pas qu'une instruction soit exécutée pour décoder la suivante. On dit que les instructions passent dans un *pipeline*, comparable à une chaîne de montage dans une usine.

Les traitements des différentes instructions se chevauchent donc temporellement, ce qui accélère l'exécution du programme par le processeur. À l'heure actuelle, le pipeline est découpé de façon suffisamment fine pour que quelques dizaines d'instructions soient simultanément en cours de traitement.

Il y a cependant une différence majeure entre une chaîne de montage et le pipeline d'un processeur. La chaîne de montage peut travailler sans arrêt, alors que les instructions de branchement perturbent considérablement le fonctionnement du pipeline.

L'architecture de von Neumann a pour avantages une certaine simplicité conceptuelle et une grande souplesse. En effet puisqu'aucune structure n'est a priori imposée sur le contenu de la mémoire, on peut stocker toute information codable numériquement. En particulier, on peut indifféremment stocker en mémoire des programmes et des données.

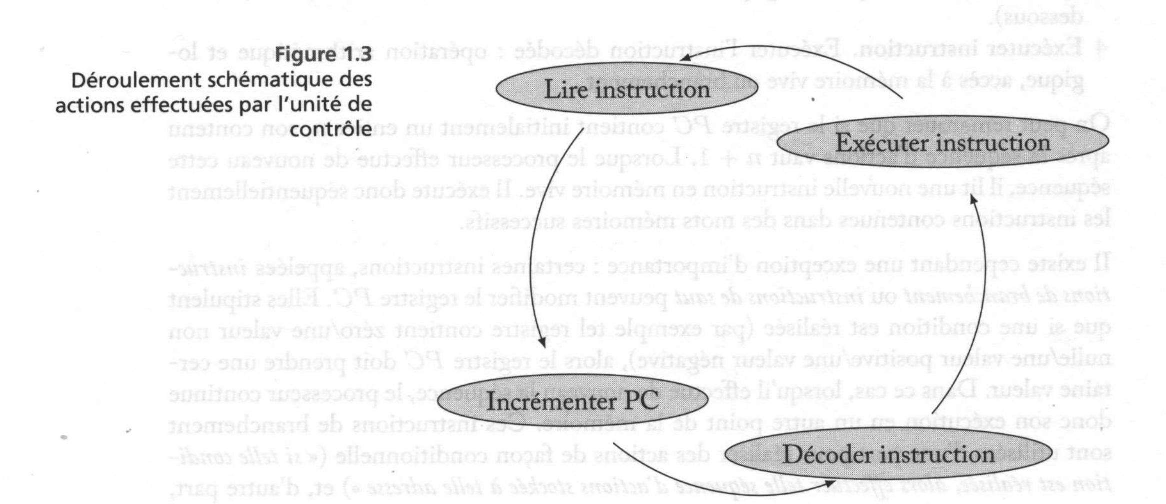


Figure 1.3
Déroulement schématique des actions effectuées par l'unité de contrôle

Elle a en revanche au moins trois gros inconvénients :

- **Une exécution séquentielle.** Les programmes sont exécutés de façon séquentielle, instruction après instruction. Une machine de von Neumann ne peut donc faire qu'une seule chose à la fois. On pourrait cependant imaginer qu'en augmentant le nombre de circuits électroniques du processeur, on pourrait effectuer plusieurs opérations en même temps. Par ailleurs, on veut que les ordinateurs soient capables d'effectuer un calcul complexe tout en décodant un fichier de musique et en le jouant, pendant que le navigateur charge une page web. On verra comment les processeurs et les systèmes d'exploitation s'affranchissent en pratique de cette limitation.
- **Un goulet d'étranglement.** Toutes les données et les instructions passent sur le bus mémoire. En pratique, les processeurs actuels sont tellement rapides que le bus a souvent bien du mal à en suivre le débit. En conséquence, le processeur passe beaucoup de temps à attendre. On exposera comment en pratique on résout ce problème.
- **Une faible robustesse.** Les données et les programmes étant mélangés, l'architecture de von Neumann est très fragile. Ainsi, il suffit qu'un bogue dans le programme conduise à écrire une donnée à l'emplacement d'un programme pour que la machine se mette à avoir un comportement inattendu, voire incompréhensible : au moment où elle lira cette donnée, elle l'interprétera comme une instruction, alors que cela n'a aucun sens.

Exercice 1.1 Avec un processeur 32 bits, combien d'adresses mémoire peuvent-elles être référencées dans un registre ? Et dans un registre 64 bits ?

En déduire pourquoi les ordinateurs de bureau actuellement en vente ont tous des processeurs 64 bits.

En 1975, Moore (un des trois fondateurs d'Intel) a prédit que le nombre de transistors des microprocesseurs doublerait tous les 2 ans. Cette loi a été relativement bien vérifiée jusqu'ici (entre 1971 et 2000, doublement tous les 1,96 années), même si on s'attend à se heurter à des limites physiques vers 2015. En conséquence, on peut estimer que la quantité de mémoire d'un ordinateur double tous les deux ans. Sous l'hypothèse (douteuse) que cette loi continue à être vérifiée, dans combien de temps les registres 64 bits seront-ils insuffisants ?

L'architecture de von Neumann décrit relativement bien les caractéristiques principales des ordinateurs actuels. Elle est cependant une architecture idéalisée. La mettre en œuvre en pratique est un peu plus délicat.

Mise en œuvre

En pratique, pour réaliser une machine suivant l'architecture de von Neumann, un gros problème se pose : comment survivre à une coupure électrique ? En effet, les mémoires vives qu'on sait aujourd'hui construire pour un coût raisonnable ont besoin d'une alimentation électrique permanente pour garder leurs données. On qualifie cette mémoire de *volatile* pour signifier qu'elle perd son contenu en cas de coupure de courant. Or, on veut que les données et programmes contenus dans un ordinateur subsistent malgré l'extinction de la machine.

Pour résoudre ce problème, on recourt à deux types de mémoires non volatiles :

- **Mémoire morte.** On sait construire des mémoires non volatiles offrant un accès en lecture à vitesse raisonnable, mais pas d'accès en écriture (*Read-Only Memory, ROM*), ou du moins qui ne peut être changée facilement (EEPROM). Ces mémoires sont en général initialisées lors de la fabrication de la machine. Elles peuvent être utilisées pour stocker des programmes mais pas les données utilisateur. Elles étaient notamment utilisées dans les consoles de jeux vidéo et certains ordinateurs des années 1980, sous la forme de cartouches enfichables dans la machine. Aujourd'hui, elles sont utilisées dans les ordinateurs de bureau pour stocker un programme qu'on appelle le *firmware* (BIOS ou UEFI). Un ordinateur de bureau possède donc à la fois une mémoire morte et une mémoire vive. Certaines plages d'adresses correspondent à des données stockées en mémoire vive, d'autres à des adresses stockées en mémoire morte et le processeur accède en lecture aux unes et aux autres indifféremment. Toute tentative d'écriture dans une adresse en mémoire morte se solde évidemment par un échec.
- **Mémoire de masse.** Pour sauvegarder les données, on ajoute un périphérique, appelé *mémoire de masse* : autrefois un lecteur de bandes magnétiques, aujourd'hui un disque dur (le plus fréquent sur les PC de bureau) ou une mémoire flash (le plus fréquent sur les smartphones). Un disque dur est capable de stocker une grande quantité de données, mais la lecture de celles-ci est environ mille fois plus lente que pour celles de la mémoire vive. De plus, l'accès à ces données est relativement complexe : comme dans le cas de la mémoire vive, les données qui y sont stockées ont toutes une adresse, mais le contenu d'une adresse donnée n'est pas accessible par une simple instruction du processeur. On stocke en général dans un disque dur non seulement les données de l'utilisateur mais aussi tous les programmes, y compris le système d'exploitation de la machine.

Par ailleurs, les constructeurs cherchent à fabriquer des ordinateurs toujours plus rapides. Pour cela, il s'agit de surmonter les problèmes inhérents à l'architecture de von Neumann. Aucune solution miracle n'existe mais plusieurs idées sont utilisées en conjonction :

- **Déléguer l'affichage.** On délègue tous les calculs concernant l'affichage à une carte graphique, sur laquelle se trouve un processeur dédié à cet usage, disposant d'une mémoire vive propre, indépendante de celle de l'ordinateur. Ce processeur est en mesure d'effectuer les calculs géométriques nécessaires pour le rendu d'une scène en trois dimensions (accélération 3D) ou pour décoder des flux vidéo, dont l'interprétation est particulièrement gourmande en calculs géométriques.
- **Laisser les périphériques accéder à la mémoire.** On permet, dans une certaine mesure, aux périphériques de recopier des données dans et depuis la mémoire (*Direct Memory Access*, DMA) pour ne pas avoir besoin de les faire passer toutes par le processeur.

En pratique, la réalisation matérielle d'un ordinateur diffère donc un peu de l'architecture idéale de von Neumann pour tenter d'en gommer les défauts.

1.2 Notion de système d'exploitation

On a décrit jusqu'ici le fonctionnement du matériel constituant un ordinateur. Ce dernier sert à exécuter un programme ; il stocke des données et les instructions du programme dans la mémoire de masse.

Or, en pratique, on n'utilise pas un mais plusieurs programmes, souvent même simultanément : navigateur web, logiciel de courrier électronique, de messagerie instantanée, traitement de texte, tableur, outil de présentation, lecteur de vidéos, jeux… Si différents outils doivent stocker des informations sur le disque dur, il convient qu'ils le fassent de façon coordonnée, afin que chacun sache où aller lire et écrire les données le concernant.

Comment la machine fait-elle pour savoir où les programmes sont stockés ? Et pour les lancer ? Comment ces programmes savent-ils ensuite où stocker leurs données ?

Répondre à ces questions est le rôle du *système d'exploitation*. Il s'agit d'un programme chargé en mémoire vive dès le démarrage de l'ordinateur et qui y reste jusqu'à l'extinction de la machine.

EN PRATIQUE Les systèmes d'exploitation

Il existe deux grandes familles de systèmes d'exploitation :

1. Les systèmes d'exploitation issus d'Unix (Mac OS X, iOS, GNU/Linux, Android, FreeBSD, NetBSD). Dans tous les domaines, sauf celui des ordinateurs personnels, les systèmes issus d'Unix sont majoritaires. Et parmi eux, Linux se taille la part du lion : depuis sa création comme un hobby d'étudiants, il y a 20 ans, Linux s'est imposé comme un système universel puisqu'il équipe aussi bien les téléphones portables (sous la forme d'Android) que les boîtiers prêtés par les fournisseurs d'accès à Internet (Freebox, Livebox, Neufbox et Bewan iBox), les ordinateurs personnels (notamment sous la forme Ubuntu/GNU Linux), les serveurs web et les supercalculateurs (plus de 90 % des calculateurs du TOP 500).
2. Les systèmes d'exploitation de la famille Microsoft Windows. Ils se sont imposés grâce aux pratiques commerciales très agressives de Microsoft et sont en situation de quasi-monopole sur les ordinateurs personnels depuis près de deux décennies (ils en équipent près de 90 %).

Le système d'exploitation a les responsabilités suivantes :

- donner l'illusion que l'ordinateur est multitâche ;
- identifier les utilisateurs ;
- gérer l'organisation du disque dur et de ses fichiers ;
- contrôler l'accès aux données du disque et de manière générale aux autres ressources de l'ordinateur ;
- gérer le lancement des différentes applications utilisées ;
- servir de garde-fou en cas de tentative de mauvaise utilisation des ressources de l'ordinateur.

1.2.1 Le multitâche

On a vu qu'un ordinateur ne peut exécuter qu'une instruction à la fois. Pourtant, le système d'exploitation permet d'exécuter plusieurs programmes en même temps. Plus exactement, il donne l'illusion que l'on exécute plusieurs programmes en même temps.

En pratique Multiprocesseurs et autres *multicore*

Certains ordinateurs exécutent réellement plusieurs instructions simultanément. Ils sont dotés de plusieurs processeurs capables d'accéder indépendamment à la mémoire et aux périphériques, ce qui revient effectivement à exécuter plusieurs instructions à la fois. Les fabricants de processeurs proposent même maintenant des processeurs qui contiennent en fait plusieurs cœurs (*dualcore, quadricore*), c'est-à-dire que plusieurs processeurs (unité arithmétique et logique et unité de contrôle) ont été regroupés en un seul. Dans tous les cas, d'une part cette intégration ne va pas sans problème pour coordonner les processeurs et, d'autre part, le nombre d'instructions exécutées simultanément est bien inférieur au nombre de programmes tournant simultanément sur l'ordinateur.

Pour donner cette illusion, le système d'exploitation stocke en mémoire les différentes applications que l'on veut exécuter. Il lance l'exécution d'une première application. Dès qu'il se produit une entrée-sortie ou, à défaut, lorsqu'un certain temps est écoulé (de l'ordre de la centaine de millisecondes), le noyau du système d'exploitation reprend la main et lance l'exécution d'une autre application. En pratique, le temps d'exécution d'une tâche dépasse rarement la dizaine de millisecondes.

De façon schématique, voici ce qui se passe lorsque l'on tape un texte sur un ordinateur tout en lui faisant jouer une fugue de Bach à l'aide d'une application de lecture audio. Le noyau du système d'exploitation commence par exemple à exécuter l'application de lecture audio, qui envoie sur le périphérique son quelques notes de la fugue. Pendant que les données sont écrites, le noyau passe la main à une autre application. Survient alors un événement : l'utilisateur vient de taper sur une touche du clavier. Le noyau reprend alors le contrôle. Il sait que le traitement de texte était en attente de cet événement : il lui passe donc la main. Celui-ci affiche la lettre tapée à l'écran et se remet en attente d'un autre caractère. Le noyau reprend donc la main pour la passer à une autre application en attente. Après quelques instants, survient alors un nouvel événement : le périphérique son signale que toutes les données ont été écrites. Le noyau passe alors de nouveau la main à l'application de lecture audio, qui envoie de nouveau quelques notes de la fugue sur le périphérique. De nouveau le noyau passe la main à une autre application. Tout cela s'est déroulé en quelques dizaines de millisecondes tout au plus.

1.2.2 Identification des utilisateurs

Les systèmes d'exploitation, Unix comme Windows, sont multi-utilisateurs : chaque utilisateur dispose d'un identifiant auprès du système (et en général, d'un mot de passe correspondant).

On prendra dans la suite de ce chapitre le cas d'un utilisateur fictif Jean Dupont utilisant un ordinateur au sein de son entreprise. Le responsable des moyens informatiques a créé un *compte* utilisateur dans le système informatique, auquel est associé un identifiant, par exemple `jdupont` et un mot de passe. Il a de plus déclaré Jean Dupont comme étant membre d'un ou plusieurs groupe(s) d'utilisateurs, ce qui lui conférera certains droits vis-à-vis du système informatique. Par exemple, si l'entreprise de Jean a défini des groupes `utilisateur`, `developpeur` et `manager` et si Jean travaille comme ingénieur informatique, on peut imaginer qu'il est membre des deux premiers groupes, mais pas du troisième. Pour des raisons qu'il serait un peu long d'expliciter ici, il est par ailleurs probable que le responsable informatique ait également créé un groupe privé `jdupont`, dont Jean sera le seul membre.

Après avoir démarré, l'ordinateur de Jean Dupont présente un *écran de connexion* (voir figure 1.5 pour un exemple). Jean inscrit alors son identifiant, puis son mot de passe. Le système d'exploitation reconnaît `jdupont` comme un identifiant valide, vérifie que le mot de passe correspond à cet identifiant et lance un programme (ou un ensemble de programmes) qu'on appelle parfois *shell*, sous l'identité `jdupont`. Sur les systèmes d'exploitation récents, ce shell se présente sous forme d'une interface graphique permettant à l'utilisateur de lancer les applications qu'il veut utiliser (navigateur web, gestionnaire de fichiers, suite bureautique, environnement de développement Python, l'application qu'il a programmée en Python...).

Il existe aussi des shells en mode texte, qu'on appelle *interprètes de commandes* : ces programmes attendent une commande de l'utilisateur sous forme d'une ligne de texte, l'exécutent, attendent de nouveau une commande, l'exécutent, etc. Ils étaient historiquement utilisés sur des terminaux en mode texte, c'est-à-dire une combinaison d'un clavier et d'un écran incapable d'afficher autre chose que du texte en orange sur fond noir (ou en vert sur fond noir). Aujourd'hui, ils ont quasiment disparu, mais tous les systèmes Unix proposent des émulateurs de terminaux. Un tel émulateur est présenté figure 1.4. Jean Dupont y a d'abord tapé les trois commandes `pwd`, `ls` et `date` qui ont eu pour effet d'afficher le nom du répertoire de travail (*print working directory*), de lister son contenu et d'afficher la date et l'heure. Il a ensuite tapé une commande plus compliquée, qui permet en une ligne, certes complexe, de savoir combien de fichiers sont présents sur l'ordinateur (on peut lire à la suite la réponse de la machine).

ATTENTION Les interprètes en mode texte

L'utilisation de ces interprètes de commandes n'a rien d'obsolète. Leur usage demande certes plus de travail au départ que l'utilisation d'un shell graphique. Pourtant, c'est l'une des façons les plus productives de faire exécuter des tâches à un ordinateur et, pour l'administrateur d'un système, c'est bien souvent un outil indispensable.

1.2.3 Système de fichiers

La mémoire de masse est généralement organisée en un *système de fichiers* qui permet aux utilisateurs d'enregistrer leurs données et leurs programmes.

Pour simplifier, on se placera ici dans le cas d'un système d'exploitation de la famille Unix. On utilisera le gestionnaire de fichiers PCManFM mais les principes énoncés restent valables pour les autres systèmes et les autres gestionnaires de fichiers.

Le nombre de fichiers est généralement très élevé : plusieurs centaines de milliers pour les programmes installés sur un ordinateur de bureau, près d'un demi-million de fichiers utilisateur accumulés en 20 ans par l'auteur de ces lignes. Ils sont organisés en une structure arborescente de répertoires. Du point de vue de l'utilisateur, un répertoire est un ensemble de fichiers et de sous-répertoires, désignés par des noms.

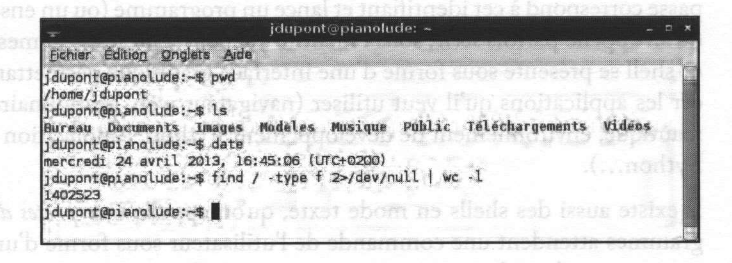


Figure 1.4 Émulateur de terminal sous un système Unix

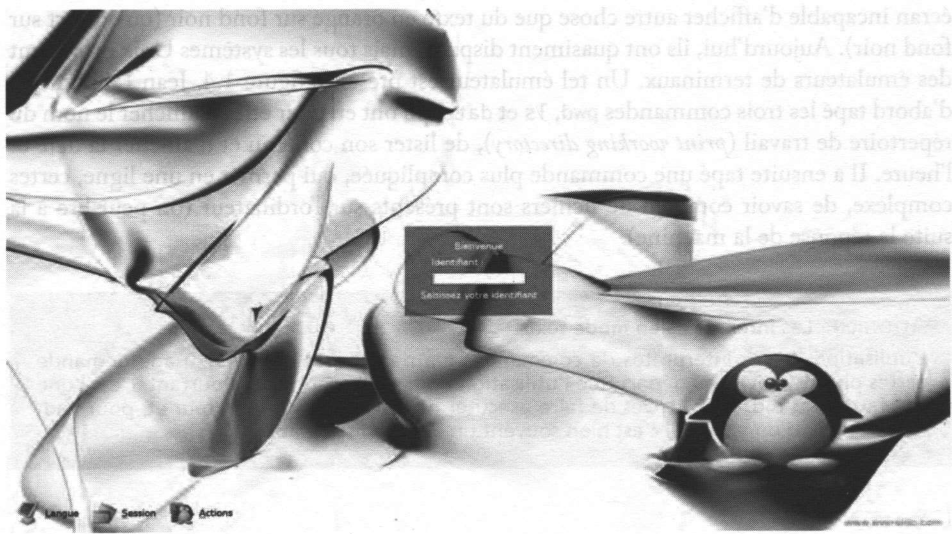
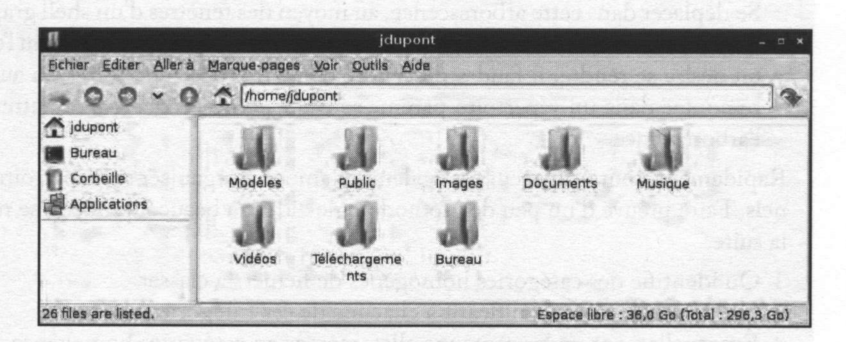


Figure 1.5 Écran de connexion

Lorsque Jean Dupont lance l'application qui explore le système de fichiers, celle-ci lui présente les fichiers d'un répertoire, en général celui où sont stockées ses propres données (voir figure 1.6).

Sous Unix, tous les fichiers sont regroupés dans une unique arborescence. Le sommet de l'arbre (qu'on appelle aussi sa racine car en informatique, les arbres poussent du haut vers le bas) est un répertoire appelé `/`. Cette racine possède plusieurs sous-répertoires, dont généralement un appelé `home`, contenant les données des utilisateurs. Dans `home`, Jean Dupont peut ainsi constater la présence d'un répertoire `jdupont`. Pour désigner un fichier (ou un sous-répertoire) `b` d'un répertoire `a`, Unix utilise la notation `a/b` (sauf si `a` est la racine, auquel cas on note simplement `/b` et non `//b`). Le répertoire de Jean Dupont, qui est le sous-répertoire `jdupont` du sous-répertoire `home` de la racine, se note donc `/home/jdupont`. C'est d'ailleurs ce qu'on peut lire dans la barre indiquant le répertoire en cours de visualisation dans le gestionnaire de fichiers (figure 1.6).

Figure 1.6
Répertoire
`/home/jdupont`



Dans les systèmes Windows, il y a quelques petites différences. Les différents disques (`C:`, `D:`, etc.) ont chacun leur propre système de fichiers, et pour séparer un nom de fichier du répertoire qui le contient on utilise la notation `a\b`. À l'intérieur de chaque disque, le principe d'organisation sous forme d'une arborescence de répertoires reste le même.

Si, dans le gestionnaire de fichiers, Jean clique sur la flèche vers le haut, il arrive dans le répertoire immédiatement au-dessus de `/home/jdupont` dans la hiérarchie, c'est-à-dire `/home`. Il constate alors que ce dernier contient d'autres sous-répertoires : `dmartin` et `xdurand` (voir figure 1.7).

Explorer toute la hiérarchie de ces fichiers est un vaste programme, mais on peut citer tout de même deux autres répertoires notables :

`/usr/bin`	Contient la plupart des applications.
`/media`	Donne accès aux périphériques de données amovibles (CD-ROM, DVD, clés USB, disques durs amovibles).

Savoir-faire Utiliser un système de fichiers

Dans un système de fichiers préexistant, il faut savoir :

- Se repérer dans l'arborescence. Cela demande de connaître dans les grandes lignes la structure du système de fichiers, de savoir retrouver son répertoire personnel et d'en connaître également l'organisation.
- Se déplacer dans cette arborescence, au moyen des fenêtres d'un shell graphique ou d'une commande (`cd` dans la plupart des systèmes d'exploitation). Selon l'endroit où on désire se rendre, il faudra descendre dans des sous-répertoires ou au contraire remonter dans un répertoire parent, souvent pour accéder à une autre partie de l'arborescence.

Rapidement, tout utilisateur est également amené à organiser ses répertoires personnels. Faire preuve d'un peu de méthodologie aidera à beaucoup mieux se repérer par la suite.

1. On identifie des catégories homogènes de fichiers à classer.
2. On donne un nom significatif à chacune de ces catégories.
3. Éventuellement on les regroupe elles-mêmes en catégories homogènes.
4. On crée les répertoires correspondant à chacune de ces catégories.
5. On met chaque fichier dans le répertoire approprié et on fait de même avec les nouveaux fichiers créés ou téléchargés par la suite.

Exercice 1.2 avec corrigé Comment organiser le répertoire personnel d'un élève qui contient aussi bien des fichiers de travail liés à ses études que des documents multimédia et des programmes de jeux ?

Typiquement, le répertoire personnel d'un élève peut contenir des fichiers entre autres dans les catégories suivantes, qui constitueront autant de répertoires : `Anglais`, `Informatique`, `Jeux`, `Musique`, `Photos`, `Sciences Physiques`.

On aura ensuite intérêt à regrouper les répertoires `Anglais`, `Informatique` *et* `Sciences Physiques` *dans un répertoire* `Travail`, *et les autres dans un répertoire* `Loisirs`.

Enfin, à l'usage, il y a des chances pour qu'un répertoire comme `Informatique` *finisse par contenir beaucoup de fichiers. On pourra donc y créer des sous-répertoires pour mieux organiser son contenu.*

POUR ALLER PLUS LOIN Comment toutes ces informations sont-elles organisées sur le disque ?

Du point de vue conceptuel, un fichier est une séquence finie d'octets, sans signification a priori : c'est le programme lisant cette séquence d'octets qui décidera de la façon de la comprendre. Il peut contenir des données de tout type : texte, son, vidéo et même des programmes directement exécutables par le processeur (programmes dits « binaires » ou « en langage machine »).

Les fichiers présents sur le disque dur sont répertoriés dans des tables, stockées à un endroit du disque conventionnellement fixé, qui donnent des informations relatives à ces fichiers, appelées métadonnées, notamment :

1. Date de création, de dernière modification, de dernière lecture.
2. Taille du fichier.
3. Emplacement des données du fichier sur le disque.
4. Suivant les systèmes de fichiers employés, un numéro identifiant le fichier, appelé *inode* sous Unix.

La table des métadonnées est aussi appelée table des inodes.

Du point de vue du système d'exploitation, un répertoire est juste un fichier particulier, qui contient une liste de couples (n, i) où n est un nom de fichier et i l'inode du fichier. Une information supplémentaire est stockée dans la table des inodes pour indiquer, pour chaque fichier, s'il contient des données ou s'il s'agit d'un répertoire.

1.2.4 Contrôle d'accès

Plutôt indiscret, Jean Dupont décide d'aller explorer les répertoires de ses collègues Dominique Martin et Xavier Durand. Cependant, lorsqu'il clique sur le répertoire xdurand, il obtient un message d'erreur, présenté figure 1.8.

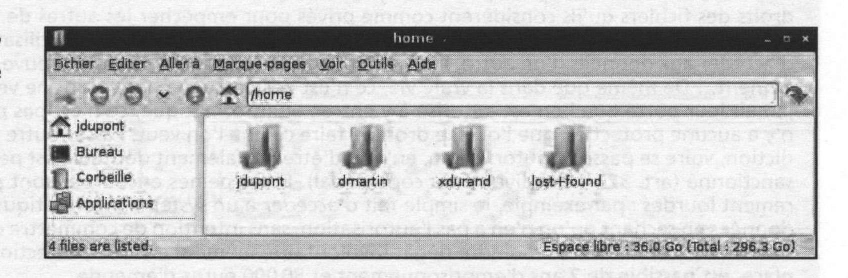


Figure 1.7
Répertoire /home

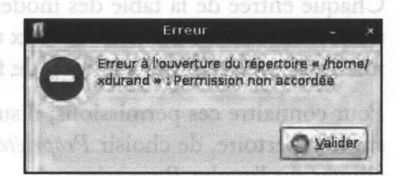


Figure 1.8
Refus d'accès à /home/xdurand

En revanche, il parvient à lire le répertoire `/home/dmartin`. Celui-ci contient un fichier `Lisez-moi.odt` que Jean Dupont parvient à ouvrir. Sous OpenOffice.org, il réussit à éditer le fichier, mais lorsqu'il essaie de l'enregistrer, il obtient un message d'erreur, reproduit figure 1.9. Qu'à cela ne tienne, Jean tente d'enregistrer ce fichier sous un nouveau nom : `nouveau-lisez-moi.odt`. Il obtient de nouveau une erreur d'OpenOffice.org (figure 1.10). Que s'est-il passé ?

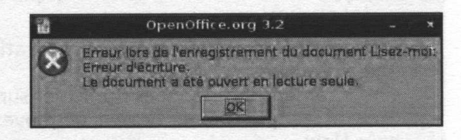


Figure 1.9
Échec de l'enregistrement d'un document protégé en écriture

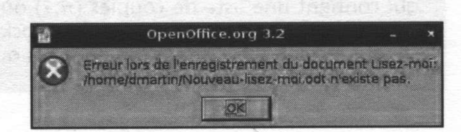


Figure 1.10
Échec de l'enregistrement d'un document dans un répertoire protégé en écriture

> ATTENTION **Les droits d'accès**
>
> Tenter d'accéder aux données d'autrui peut être considéré comme une atteinte à la vie privée. Dans certaines entreprises, on fait l'hypothèse que tous les utilisateurs savent gérer les droits des fichiers qu'ils considèrent comme privés pour empêcher les autres de les lire ou de les modifier. Dans d'autres, on considérera qu'il n'est pas normal qu'un utilisateur tente d'accéder aux données d'un autre. Il convient de se renseigner et de faire preuve de discernement… De même que dans la vraie vie, ce n'est pas parce que les voisins ne verrouillent jamais leur porte que l'on est autorisé à y entrer, en informatique, ce n'est pas parce qu'il n'y a aucune protection que l'on a le droit de faire ce que l'on veut. Passer outre une interdiction, voire se passer d'autorisation, en plus d'être moralement douteux, est pénalement sanctionné (art. 323-1 et suivants du code pénal). Et les peines encourues sont particulièrement lourdes : par exemple, le simple fait d'*accéder* à un système informatique ou à des données en sachant qu'on n'en a pas l'autorisation, sans intention de commettre de dégâts, sans même commettre le moindre dégât involontaire, même si aucune protection n'est en place, est passible de 2 ans d'emprisonnement et 30 000 euros d'amende.

Chaque entrée de la table des inodes comporte, en plus des métadonnées déjà mentionnées, les droits d'accès accordés aux utilisateurs du système. Ces droits, appelés aussi permissions, précisent qui a le droit de faire quoi sur le fichier ou le répertoire concerné.

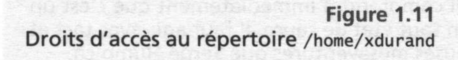
Pour connaître ces permissions, il suffit, dans le gestionnaire de fichiers, de cliquer-droit sur le répertoire, de choisir *Propriétés* dans le menu contextuel qui apparaît alors, puis de cliquer sur l'onglet *Permissions*.

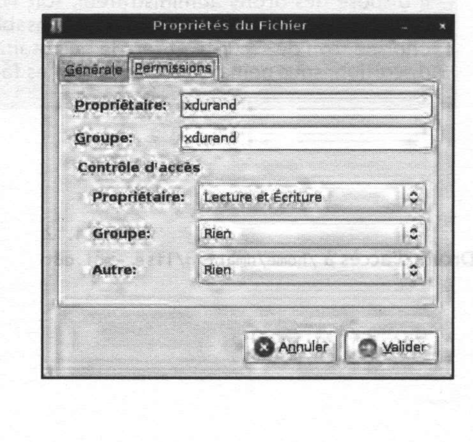
Lorsque Jean Dupont regarde ainsi les permissions de `/home/xdurand` (voir figure 1.11), il constate que les droits sont les suivants :

- L'utilisateur propriétaire du fichier est `xdurand`.
- Le groupe auquel appartient le fichier est aussi `xdurand`.
- Le propriétaire du fichier peut accéder en lecture et en écriture au fichier.
- Les autres utilisateurs n'ont aucun droit sur le fichier.

Lorsqu'un programme, quel qu'il soit, tente d'accéder à un fichier ou à un répertoire, il ne peut le faire directement : il doit demander les données qui l'intéressent au système d'exploitation. Celui-ci regarde alors sous quelle identité tourne le programme et quels sont les droits d'accès du fichier concerné. Si les permissions de l'utilisateur sont insuffisantes pour l'opération demandée, le système refuse. D'où les messages d'erreurs constatés :

- Dans le cas présenté figure 1.9, Jean Dupont essaie d'écrire dans un *fichier* sur lequel il n'a pas le droit d'écriture, comme on peut le constater figure 1.12. OpenOffice.org l'a constaté et a adapté son message d'erreur.
- Dans le cas présenté figure 1.10, Jean essaie d'écrire le fichier en lui donnant un nouveau nom, `nouveau-lisez-moi.odt` dans un *répertoire* (`/home/dmartin`) sur lequel il n'a pas le droit d'écriture. OpenOffice.org n'a pas cherché à vérifier si cela était possible et a demandé au système de créer le fichier. Il n'a probablement pas vérifié la réponse du système d'exploitation à sa requête[1] puis découvre au moment d'écrire dans le fichier que celui-ci n'existe pas, d'où le message d'erreur quelque peu sibyllin.


Figure 1.11
Droits d'accès au répertoire `/home/xdurand`



1. Il y a lieu de penser qu'il s'agit d'un bogue d'OpenOffice.org

Plus généralement, le système d'exploitation contrôle l'accès aux différentes ressources de l'ordinateur :

Écran, carte son, clavier, souris	Les utilisateurs d'un ordinateur ne sont pas nécessairement tous face à l'ordinateur. Certains peuvent être connectés à distance. Les droits d'accès à la carte son et à l'écran sont en général accordés à l'utilisateur physiquement présent devant la console (ou plus exactement, à celui dont l'identifiant et le mot de passe ont été entrés par l'intermédiaire du clavier). La lecture des informations de la souris et du clavier est réservée également à cet utilisateur.
Imprimante	Selon les choix faits par l'administrateur du système, l'usage de l'imprimante peut être réservé aux utilisateurs physiquement présents ou au contraire ouvert à tous les utilisateurs.
Réseau	De même, l'accès au réseau peut être ouvert à tous ou filtré par utilisateur ou par contenu.

Exercice 1.3 Sur un PC auquel vous avez accès, en tant que simple utilisateur :

1 Essayez de copier un de vos fichiers dans un répertoire du système (par exemple, `/etc` sous Unix ou `C:\Program Files` sous MS-Windows). Que se passe t-il ?

2 Sous Linux seulement, essayez d'ouvrir le fichier `/etc/shadow`. Que se passe t-il ?

ATTENTION Les droits administrateur

Il devrait être impossible de copier ou d'ouvrir le fichier `/etc/shadow`. Aucun fichier ne devrait pouvoir être créé dans `/etc` ou `C:\Program Files`. Si un utilisateur y parvient, c'est que soit il dispose des droits administrateur, soit l'ordinateur est mal configuré. Dans ce dernier cas, il faut prévenir d'urgence le responsable. Il comprendra immédiatement que c'est un énorme trou de sécurité et fera le nécessaire. En tout état de cause, il faut agir avec tact et discrétion, sous peine de vivre les mêmes fâcheuses mésaventures que Serge Humpich.

Figure 1.12
Droits d'accès à `/home/dmartin/lisez-moi.odt`

1.2.5 Lancement d'applications

SAVOIR-FAIRE Lancer des applications

Pour lancer une application (un programme), on dispose de deux possibilités :

- **Cliquer sur un bouton, un menu ou un fichier du shell graphique.** Cette méthode est en général à réserver pour des tâches simples ; elle offre l'avantage d'être intuitive et donc utilisable même par des utilisateurs débutants.
- **Taper une commande dans un shell en mode texte.** Dès que l'on veut exprimer des commandes plus complexes, par exemple passer des arguments à un programme ou manipuler des fichiers en masse, cette méthode s'avère quasi incontournable. Son utilisation est plus ou moins facilitée selon le système d'exploitation que l'on utilise.

Le shell graphique rend transparentes pour l'utilisateur les étapes du lancement d'une application.

Pour démarrer par exemple OpenOffice.org, l'utilisateur Jean Dupont clique sur un bouton pour faire apparaître un menu dans lequel il choisit l'application. Le shell graphique « sait » que le programme OpenOffice.org est installé. Plus précisément, il sait qu'un certain fichier (nommé `ooffice`) est exécutable par le système. Il demande donc au système d'exploitation d'exécuter ce programme. Ensuite, tout se passe comme si l'utilisateur avait tapé directement la commande `ooffice` dans un shell en mode texte.

Le système commence par vérifier que l'utilisateur a le droit d'exécuter le programme. Sur la figure 1.12, on constate en effet la présence d'une case *Rendre le fichier exécutable*. Elle correspond à une permission d'exécution attachée au fichier, au même titre que les permissions de lecture et d'écriture.

Ensuite, le système réserve un espace dans la mémoire vive de l'ordinateur pour stocker les instructions du programme, ainsi que ses données. Il copie le contenu du fichier exécutable en mémoire. Celui-ci n'est qu'une suite de bits qui codent les instructions dans le langage du processeur (on dit que le programme est en langage machine), il peut donc les exécuter en effectuant un branchement vers les premières instructions du programme.

Du point de vue de l'utilisateur, il existe d'autres façons de lancer un programme, mais qui restent à rapprocher du shell graphique. Par exemple, si Jean Dupont clique dans son gestionnaire de fichiers sur un fichier OpenDocument, celui-ci va lancer OpenOffice.org. En effet, lorsqu'on clique sur le nom ou l'icône d'un fichier, le gestionnaire de fichiers commence par en déterminer le type.

Suivant les systèmes, il peut s'appuyer sur diverses informations :

- le nom du fichier et tout particulièrement son suffixe. Ainsi, un fichier dont le nom se termine par `.odt` sera t-il identifié comme un fichier OpenDocument (le gestionnaire consulte pour cela une table qui associe un type de fichier à chaque suffixe connu) ;
- le contenu du fichier ;
- le fait que l'utilisateur ait ou non le droit d'exécuter ce fichier ;
- sur certains systèmes (Mac OS X en particulier), les métadonnées associées au fichier donnant son type.

Une fois le type déterminé, le gestionnaire de fichiers consulte une table (commune à tout le système ou spécifique au gestionnaire, commune à tous les utilisateurs ou spécifique à l'utilisateur) indiquant à quelle application est associé ce type de fichier. Il lance alors l'application, en lui fournissant pour *argument* (c'est-à-dire comme information supplémentaire) le nom du fichier à ouvrir. Là encore, la situation est exactement la même que si on avait tapé `ooffice` *`nomdufichier`* dans un shell en mode texte.

SAVOIR-FAIRE **Écrire un programme et le lancer**

Tout utilisateur peut écrire ses propres programmes. Pour cela, il a trois possibilités :

- **Les écrire en langage machine.** C'est une tâche ardue car il s'agit d'un langage de bas niveau dans lequel il est difficile ou au minimum fastidieux d'implanter des idées un tant soit peu complexes. De plus un tel programme est spécifique à la machine pour laquelle il a été écrit et risque de ne pas fonctionner sur une autre.
- **Utiliser un *compilateur*.** Un programme écrit dans un langage évolué est *traduit* par le compilateur pour donner un programme en langage machine.
- **Faire appel à un interpréteur d'un langage évolué.** Un interpréteur est un programme exécutable qui va lire le texte d'un programme dans un langage évolué pour l'exécuter pas à pas, sans passer par la phase intermédiaire de compilation.

La compilation demande un traitement préliminaire avant de pouvoir exécuter le programme que l'on a écrit, mais produit en général des applications efficaces. À l'inverse, on peut interpréter un programme immédiatement après l'avoir écrit, mais il aura tendance à s'exécuter moins rapidement.

Le choix entre compilation et interprétation dépend très largement du langage dans lequel on programme : certains ne proposent qu'une seule des deux possibilités, d'autres laissent le choix. Il existe même des situations intermédiaires où un programme peut être compilé dans un langage qui est plus proche du langage machine mais qui doit encore être interprété (on parle de *bytecode*). Pour des applications écrites en Python, à l'heure actuelle, on utilise usuellement un interpréteur. Voir en annexe le TP sur la création de programmes autonomes, section A.1 pour la mise en œuvre concrète de la compilation ou de l'interprétation d'un programme.

1.2.6 Protections

Jean Dupont, désirant toujours lire des données qui lui sont interdites, pourrait imaginer écrire son propre programme, qui accéderait directement au disque dur pour ne pas avoir à demander la permission au système d'exploitation. Après tout, puisqu'il est possible de faire exécuter au processeur des instructions arbitraires, ne pourrait-il pas en tirer parti ?

Cela n'est pas si simple : lorsqu'il lance un programme utilisateur, le système d'exploitation fait basculer le processeur dans un mode de fonctionnement particulier (appelé *mode utilisateur*) dans lequel le programme est isolé : il ne peut accéder ni à l'espace mémoire des autres processeurs, ni aux périphériques du système.

S'il tente malgré tout de lire ou d'écrire dans des parties de la mémoire qui lui sont interdites, son exécution est momentanément arrêtée et le contrôle est transféré au noyau du système d'exploitation, pour qu'il décide de la conduite à tenir. Sa décision est (normalement) sans appel : le programme est définitivement arrêté[2].

Pour ce qui est de l'accès aux périphériques, la seule possibilité pour le programme consiste à appeler des fonctions du noyau du système d'exploitation (on parle d'*appels système*). On peut les voir comme des branchements d'un type particulier, qui transfèrent le contrôle au système d'exploitation et remettent le processeur en *mode noyau*. Ensuite, le système d'exploitation fait parvenir sa réponse au programme et lui rend le contrôle en mode utilisateur.

Un programme lancé par Jean Dupont doit donc passer par un appel système pour accéder à un fichier. Comme on l'a vu, le système vérifiera les droits de l'utilisateur du programme et refusera si celui-ci n'est pas autorisé à y accéder.

Ainsi le système d'exploitation protège-t-il les différents programmes et utilisateurs les uns des autres.

1.3 Environnement de développement intégré

Un ordinateur est donc une machine universelle, sur laquelle tout utilisateur a la capacité d'écrire et d'exécuter un programme, dans les limites fixées par le système d'exploitation. Il serait cependant assez ardu de produire des programmes complexes s'il n'existait pas des moyens adaptés pour les développer. Cette section présente une famille de logiciels qui ont pour seul but de faciliter la conception de programmes, en fournissant dans un cadre unique la plupart des outils dont peut avoir besoin un programmeur.

2. Sous MS-Windows, on peut voir apparaître des fenêtres d'erreur indiquant « Ce programme va être arrêté parce qu'il a effectué une opération non conforme ». Il s'agit de la réaction du système d'exploitation à une erreur de ce type.

On appelle *environnement de développement intégré* (parfois abrégé IDE) un logiciel qui permet à la fois :

- d'*écrire* des programmes dans un éditeur adapté au langage ;
- d'*exécuter* les programmes que l'on a écrits ;
- de corriger des erreurs (*déboguer*) dans ces programmes ;
- éventuellement de consulter de la documentation.

Il existe de nombreux logiciels de ce type, chacun ayant ses particularités : spécifique à un langage de programmation ou générique, léger ou complet... Cependant, tous fonctionnent de façon similaire : l'essentiel est de comprendre les principes généraux et de choisir un IDE avec lequel on travaille confortablement. Les outils cités ci-après à titre d'exemple sont tous gratuits et fonctionnent sur le principe du logiciel libre ; il existe aussi des IDE commerciaux.

- **IDLE.** Fourni avec la distribution standard de Python, il est particulièrement sobre, donc suffisant pour une utilisation basique de Python.
- **Eclipse.** Plus lourd à manipuler, il possède des plugins pour à peu près n'importe quel langage de programmation (dont PyDev pour Python) et pourra donc intéresser les étudiants qui souhaiteraient utiliser le même IDE dans différents contextes.
- **Emacs ou Vim.** Ce sont des éditeurs de texte qui peuvent être étendus pour exécuter des programmes Python ou même d'autres langages. Si cette solution a l'avantage d'être particulièrement légère, elle est évidemment très dépouillée et n'est pas forcément facile à prendre en main.

Cet ouvrage montrera comment utiliser Spyder [3], qui est fourni avec plusieurs distributions de Python : WinPython ou Python(x,y) sous Windows, via le projet MacPorts sous Mac OS, et enfin dans différents paquets pour la plupart des distributions Linux. Le lecteur n'aura cependant aucun mal à retrouver les mêmes fonctionnalités dans d'autres environnements.

L'avantage le plus significatif de Spyder par rapport aux autres distributions est que les bibliothèques utilisées en classes préparatoires sont fournies directement, ce qui en simplifie largement l'installation.

La fenêtre de Spyder est divisée en trois parties (figure 1.13) :

- **L'éditeur** à gauche, dans lequel on écrira les programmes.
- **L'explorateur** en haut à droite, que nous utiliserons surtout comme débogueur, mais qui peut également servir de documentation.
- **La console interactive** en bas à droite, dans laquelle s'exécuteront les programmes.

3. À l'heure où cet ouvrage est imprimé, Spyder en est à sa version 2.2 et utilise exclusivement Python 2.x. La version 2.3, qui devrait être disponible sous peu et permettre d'utiliser Python 3.x, est déjà accessible à titre expérimental.

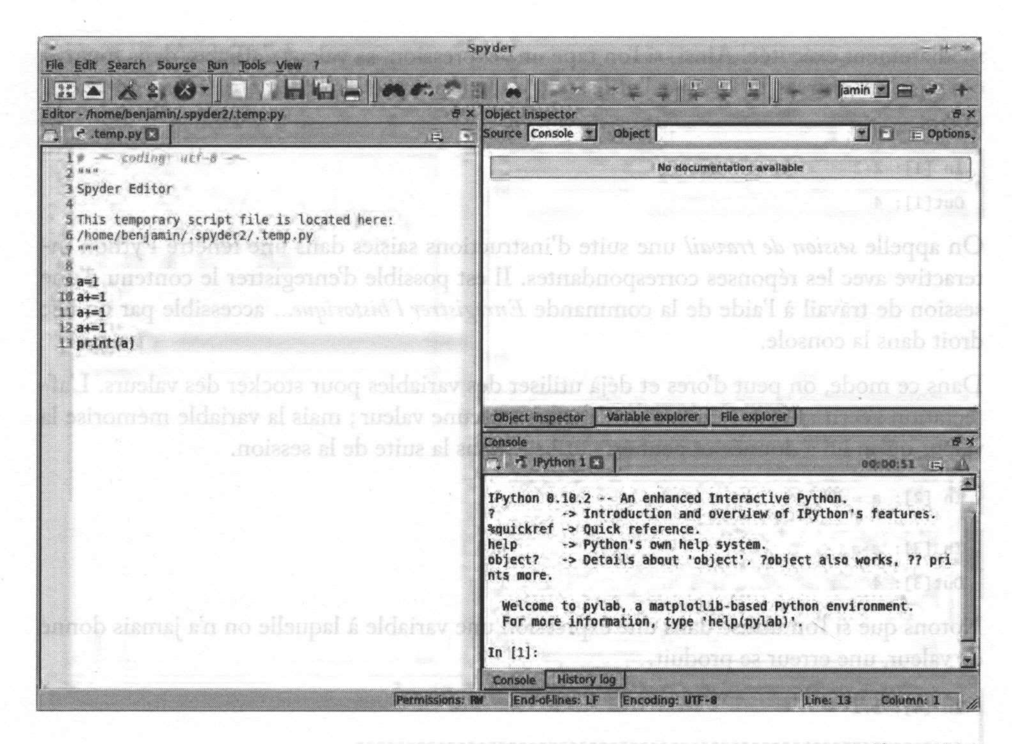


Figure 1.13
Le logiciel Spyder au démarrage

1.3.1 Console interactive

Au démarrage de Spyder, la console interactive affiche des informations sur la version de Python utilisée, ainsi que quelques fonctions d'aide :

```
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.10.2 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'.?object also works,?? prints more.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [1]:
```

La dernière ligne, qui commence par `In` suivi d'un nombre entre crochets, attend que l'on tape une commande : c'est le mode *interactif* de Python, où chaque ligne tapée est im-

médiatement exécutée. Ainsi, si l'on tape une expression, sa valeur s'affiche (dans tout cet ouvrage, les entrées tapées par l'utilisateur dans l'interpréteur interactif seront notées en gras) :

```
In [1]: 2+2
Out[1]: 4
```

On appelle *session de travail* une suite d'instructions saisies dans une fenêtre Python interactive avec les réponses correspondantes. Il est possible d'enregistrer le contenu d'une session de travail à l'aide de la commande *Enregistrer l'historique...* accessible par un clic droit dans la console.

Dans ce mode, on peut d'ores et déjà utiliser des variables pour stocker des valeurs. L'affectation s'écrit avec le symbole = et n'affiche aucune valeur ; mais la variable mémorise la valeur qu'on lui a donnée et peut être utilisée dans la suite de la session.

```
In [2]: a = 2
```

```
In [3]: a+a
Out[3]: 4
```

Notons que si l'on utilise dans une expression une variable à laquelle on n'a jamais donné de valeur, une erreur se produit.

```
In [4]: b+1
---------------------------------------------------------------
Traceback (most recent call last):
 File "<ipython console>", line 1, in <module>
NameError: name 'b' is not defined
```

La dernière ligne de ce message indique plus précisément d'où vient l'erreur, ici de la variable *b* utilisée à tort. Dans la suite de cet ouvrage, seule cette ligne sera reproduite pour expliquer une erreur.

```
In [5]: b = 1
```

```
In [6]: b+1
Out[6]: 2
```

Enfin, il est possible de rappeler une ligne tapée précédemment à l'aide des flèches *haut* et *bas*, et de modifier cette ligne avant de relancer son calcul avec *Entrée*.

```
In [7]: b = b+1
```

```
In [8]: b = b+1
```

```
In [9]: b = b+1
```

```
In [10]: b
Out[10]: 4
```

À chaque nouvelle session, les valeurs des variables sont perdues ; les instructions précédemment saisies peuvent toujours être rappelées, mais cela reste peu pratique et on n'imagine évidemment pas écrire un programme complet de la sorte. Les sessions interactives sont donc à réserver pour tester très rapidement l'évaluation de quelques expressions que l'on ne souhaite pas conserver par la suite.

1.3.2 Éditeur

Dès que l'on veut écrire un programme, ou même tout simplement une suite d'instructions dont on veut garder une trace, on utilise l'éditeur.

Voici un premier programme Python à tester :

```
print("Bonjour !")
x = 42
print(x)
```

On observe déjà plusieurs différences par rapport au mode interactif :

- Les mots-clés du langage (comme **print**) et les nombres se colorent pour ressortir sur le reste du texte.
- Les chaînes de caractères (entre guillemets) se colorent également.
- Lorsque l'on tape une parenthèse ouvrante, la parenthèse fermante correspondante se crée automatiquement ; et lorsque l'on place le curseur à droite d'une parenthèse, celle-ci se colore en rouge s'il manque la parenthèse correspondante, en vert sinon.

Ainsi, le programme écrit devient plus lisible et on évite de nombreuses fautes de frappe.

Cependant, à ce stade, le programme n'est encore qu'un texte, une suite de caractères qui n'a pas de sens pour l'ordinateur. Pour que la machine exécute (on dit aussi *interprète*) les instructions que l'on a tapées, il faut le lui demander par la commande *Exécution* du menu du même nom (raccourci clavier *F5* ou icône ▶). Les instructions sont alors lues et exécutées ; le résultat, lorsqu'il y en a un, s'affiche dans l'interpréteur interactif.

On précise qu'il est possible, au moyen de la commande *Configurer...* du menu *Exécution* (✗), de choisir si on utilise une nouvelle fenêtre interactive ou bien celle déjà manipulée. L'intérêt d'une telle option est que, tant qu'une session interactive reste ouverte, elle garde la mémoire des variables auxquelles on a affecté des valeurs. Cela peut influer sur d'autres instructions (interactives ou non) que l'on exécuterait par la suite ; cela permet par ailleurs de consulter la valeur des variables à la fin de l'exécution d'un programme.

SAVOIR-FAIRE Utiliser un environnement de développement intégré

Le minimum pour utiliser un environnement de développement intégré est de savoir :
1 Lancer l'IDE.
2 Ouvrir et enregistrer les programmes que l'on écrit.
3 Exécuter ces programmes.

L'IDE est généralement accessible au moyen d'un raccourci du système d'exploitation, ou en tapant son nom dans un shell texte. La manipulation des programmes se fait par le biais des menus de l'IDE ou par des raccourcis clavier, qui s'avèrent plus efficaces lorsqu'on en a pris l'habitude.

Enfin, l'utilisation d'un IDE prend tout son sens lorsque l'on se sert de son *débogueur*.

1.3.3 Débogueur

La fenêtre située en haut à droite de Spyder possède plusieurs onglets :

- **L'inspecteur d'objets** fournit de l'aide sur un type ou sur une fonction.
- **L'explorateur de variables** indique la valeur de toutes les variables à tout moment.
- **L'explorateur de fichiers** parcourt les fichiers Python du système de fichiers.

Celui qui va servir plus particulièrement est l'onglet *Explorateur de variables*. Par exemple, après l'exécution du programme écrit plus haut, on retrouve dans l'explorateur la variable `x` avec une valeur de `42`. Si l'interpréteur interactif dans lequel on travaille est le même depuis le début, on y retrouvera également les variables `a` et `b` que l'on a affectées directement dans la console. Enfin, l'explorateur mentionne parfois d'autres valeurs qui ont été initialisées au lancement de l'interpréteur, comme les constantes `e` ou `pi`.

Pour chercher une erreur dans un programme, ou tout simplement pour mieux comprendre son fonctionnement, il peut être utile de l'exécuter pas à pas. Si cela reste faisable de tête pour des programmes simples, il devient vite beaucoup plus commode de confier cette tâche à la machine.

On prendra l'exemple suivant :

```
x = 10
y = 7
x = x+y
y = x
x = 5 / (x-y)
```

Après avoir recopié ce programme dans l'éditeur, au lieu de l'exécuter normalement, on appelle la commande *Déboguer* du menu *Exécution* (raccourci `Ctrl+F5` ou 🐞). La première ligne du programme (`x = 10`) est surlignée et recopiée dans l'interpréteur, mais elle n'est pas encore exécutée : on peut le vérifier en constatant que `x` n'a pas la valeur `10` dans l'explorateur

de variables. Pour exécuter cette ligne, on peut cliquer sur le bouton *Pas en avant* , ou bien taper `n` (comme `next`) dans l'interpréteur. La ligne est alors exécutée, `x` prend la valeur `10` et c'est la ligne suivante (`y = 7`) qui est surlignée. On peut ensuite continuer à exécuter les lignes les unes après les autres et surveiller l'évolution des différentes variables dans l'explorateur.

Comme il est malcommode de cliquer sur un bouton pour chaque ligne alors que seules certaines posent problème, on peut fixer des *points d'arrêts* dans le programme. Pour cela, on se place sur la ligne à laquelle on désire faire une pause et on choisit *Ajouter un point d'arrêt* dans le menu *Exécution* (raccourci *F12*). On peut également double-cliquer dans la marge gauche du programme ; le symbole s'affiche alors en face de cette ligne. Par exemple, dans le programme précédent, il peut être judicieux de placer un point d'arrêt à la dernière ligne, dans laquelle risque de se produire une division par zéro.

Désormais, si au lieu d'effectuer *Pas en avant*, on choisit *Continuer* (▶ ou `c` dans l'interpréteur interactif), le programme s'exécute normalement jusqu'au prochain point d'arrêt, puis attend un ordre de l'utilisateur pour continuer à s'exécuter. Dans cet exemple, on voit dans l'explorateur de variables que `x` et `y` contiennent toutes deux la valeur `17` et qu'il y aura donc effectivement une division par zéro.

SAVOIR-FAIRE Utiliser un débogueur

Pour corriger une erreur dans un programme :

- on identifie les variables qui ne se comportent pas comme prévu ;
- on localise l'erreur au moyen d'une exécution pas à pas.

Dans le cas d'un programme de taille trop importante pour une exécution pas à pas complète, on veillera à placer des points d'arrêt aux endroits critiques, c'est-à-dire sur les opérations arithmétiques pouvant produire des erreurs, les fins de boucles, certains tests…

Exercice 1.4 Cet exercice a pour seul but de s'entraîner à manipuler l'environnement de développement intégré, il n'est pas nécessaire de savoir déjà programmer en Python pour le réaliser.

1 Taper le programme suivant dans l'éditeur et l'exécuter. Que se passe-t-il ?

```
i = 10
while i != 0:
    i = 1-i
    print(i)
```

(Il est possible qu'à ce stade il soit nécessaire de fermer complètement l'IDE et de le rouvrir pour continuer…).

2 Exécuter ce programme pas à pas et observer les valeurs successives prises par la variable `i`. Expliquer le comportement observé à la première question.

3 Placer un point d'arrêt à un endroit approprié du programme pour montrer son comportement sans avoir besoin de détailler des étapes inutiles.

2
Représentation des nombres

Les ordinateurs et les programmes mémorisent, transmettent et transforment des données aussi variées que des nombres, des textes, des images, des sons, etc. Pourtant, quand on les observe à une plus petite échelle, les ordinateurs ne manipulent que des objets beaucoup plus simples : des 0 et des 1. Dans ce chapitre, nous verrons comment cela suffit à représenter la plupart des nombres (des entiers naturels aux nombres à virgule).

2.1 Représentation des entiers naturels

2.1.1 Représentation de l'information par des booléens

La mémoire des ordinateurs est constituée d'une multitude de petits circuits électroniques qui, chacun, ne peuvent être que dans deux états : sous tension ou hors tension. Techniquement, il serait possible de construire des circuits ayant plus de deux états, correspondant à différentes valeurs de tensions, mais les risques d'erreurs dans le stockage et la transmission de ces états deviennent vite beaucoup plus importants que les avantages qu'on pourrait en tirer. Comme il fallait donner un nom à ces états, on a décidé de les appeler 0 et 1, mais on aurait pu tout aussi bien les appeler A et B, froid et chaud, faux et vrai, etc. Une telle valeur, 0 ou 1, est dite booléenne. On l'appellera booléen, chiffre binaire ou encore *bit* (*binary digit*). Un tel circuit à deux états s'appelle un circuit mémoire un bit.

L'état dans lequel se trouve un circuit mémoire un bit est représenté par le symbole 0 ou par le symbole 1. L'état d'un circuit composé de plusieurs de ces circuits est représenté par une suite finie de 0 et de 1, que l'on appelle un mot. Par exemple, le mot 100 décrit l'état d'un circuit composé de trois circuits mémoire un bit, respectivement dans l'état 1, 0 et 0.

Exercice 2.1 Trouvez trois informations de la vie courante qui peuvent être exprimées par un booléen.

Exercice 2.2 On imagine un ordinateur dont la mémoire est constituée de quatre circuits mémoire un bit. Quel est le nombre d'états possibles de la mémoire de cet ordinateur ? Même question pour un ordinateur dont la mémoire est constituée de dix circuits mémoire un bit. Et pour un ordinateur dont la mémoire est constituée de 32 milliards de tels circuits ?

Exercice 2.3 On veut représenter chacune des sept couleurs de l'arc-en-ciel par un mot, les sept mots devant être distincts et de même longueur. Quelle est la longueur minimale de ces mots ?

Exercice 2.4 Un horloger excentrique a eu l'idée de fabriquer une montre sur laquelle l'heure est indiquée par dix diodes électroluminescentes appelées 1 h, 2 h, 4 h, 8 h, 1 min, 2 min, 4 min, 8 min, 16 min et 32 min. Pour connaître l'heure, il suffit d'ajouter la valeur des diodes allumées.
Quelle heure est-il quand sont allumées les diodes 1 h, 2 h, 4 h, 1 min, 2 min, 8 min, 16 min et 32 min ? Quelles sont les diodes allumées à 5 h 55 ? Est-il possible de représenter toutes les heures ? Toutes les configurations sont-elles la représentation d'une heure ?

2.1.2 La numération à position et les bases

Depuis le Moyen-Âge, on écrit les nombres entiers naturels en notation décimale à position. Cela signifie que, pour écrire le nombre entier naturel n, on commence par imaginer n objets, que l'on groupe par paquets de dix, puis on groupe ces paquets de dix objets en paquets de dix paquets, etc. À la fin, il reste entre zéro et neuf objets isolés, entre zéro et neuf paquets isolés de dix objets, entre zéro et neuf paquets isolés de cent objets, etc. Et on représente cet entier naturel en écrivant de droite à gauche, le nombre d'objets isolés, le nombre de paquets de dix, le nombre de paquets de cent, le nombre de paquets de mille, etc. Chacun de ces nombres étant compris entre zéro et neuf, seuls dix chiffres sont nécessaires : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. Par exemple, l'écriture 2359 exprime un entier naturel formé de 9 unités, 5 dizaines, 3 centaines et 2 milliers.

Le choix de faire des paquets de dix est conventionnel : on aurait tout aussi bien pu décider de faire des paquets de deux, de cinq, de douze, de vingt, de soixante, etc. On écrirait alors les nombres entiers naturels en notation à position en base deux, cinq, douze, vingt ou soixante. La notation décimale s'appelle donc aussi notation à position en base dix.

EN PRATIQUE Les bases en informatique

Comme on le verra par la suite, la base deux joue un rôle prépondérant en informatique, en raison de l'organisation de la mémoire sous forme de suites de bits. L'autre base également utilisée de façon fréquente est la base 16, dite *hexadécimale*. Elle permet de décrire ou de manipuler une longue suite de bits sans lui donner de signification particulière. C'est utile pour créer une clé de connexion à un réseau WiFi ou une adresse mémoire. Il devient vite malcommode de le faire en base deux et on multiplie les risques d'erreurs alors que ce sont justement des données sur lesquelles une erreur est fatale. On regroupe donc ces bits par paquets de quatre, ce qui donne des valeurs en base $2^4 = 16$ (voir plus loin l'exercice corrigé 2.7 pour leur écriture). L'écriture obtenue est plus compacte (4 fois moins de chiffres pour un même nombre) mais reste manipulable. Une autre solution aurait été de regrouper les bits par paquets de 8, comme on le fait dans les octets ; mais il aurait alors fallu $2^8 = 256$ chiffres différents, et l'alphabet n'y suffirait plus. Un octet sera donc représenté par deux chiffres hexadécimaux.

Dans ce livre, quand une suite de chiffres exprime un nombre dans une base différente de dix, on indique la base en indice, par exemple : 1101_2. On souligne aussi parfois un mot pour indiquer qu'il exprime un nombre en base deux : $\underline{1101}$. Enfin, on rassemble parfois les bits par groupes de quatre ou de huit dans les mots très longs pour qu'ils soient plus faciles à lire : 1111111101 est écrit 11 1111 1101. Comme en base dix, ces groupes sont formés de droite à gauche.

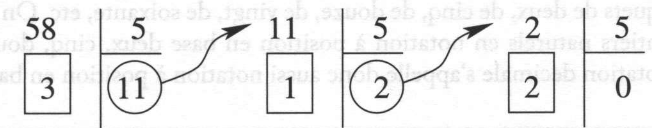
SAVOIR-FAIRE Représenter en base k un entier naturel donné en base dix

Pour écrire les entiers naturels en base k, on a besoin de k chiffres. Quand on a n objets, on les groupe par paquets de k, puis on groupe ces paquets en paquets de k paquets, etc. Autrement dit, on fait une succession de divisions par k, jusqu'à obtenir un quotient égal à 0.

Plus formellement, si l'on note $a_i \dots a_1 a_0$ l'écriture de n en base k, elle est obtenue par l'algorithme suivant :

Données : n, k
$i \leftarrow 0$
tant que $n \neq 0$ **faire**
  $a_i \leftarrow$ reste de la division euclidienne de n par k
  $n \leftarrow$ quotient de la division euclidienne de n par k
  $i \leftarrow i + 1$
Résultat : la représentation $a_i \dots a_1 a_0$.

Exercice 2.5 avec corrigé Trouver la représentation en base cinq de 58.



Donc, 58 objets se regroupent en 11 paquets et 3 unités, puis les 11 paquets se regroupent en 2 paquets de paquets et 1 paquet.

$$58 = 11 \times 5 + \mathbf{3} = (2 \times 5 + \mathbf{1}) \times 5 + \mathbf{3} = (\mathbf{2} \times 5^2) + (\mathbf{1} \times 5^1) + (\mathbf{3} \times 5^0)$$

Donc $58 = 213_5$.

Exercice 2.6 avec corrigé Trouver la représentation en base cinq du nombre 872.

872 | 5 → 2 | 174 ; 174 | 5 → 4 | 34 ; 34 | 5 → 4 | 6 ; 6 | 5 → 1 | 1 ; 1 | 5 → 1 | 0

Donc $872 = 11442_5$.

Exercice 2.7 avec corrigé Trouver la représentation en base seize du nombre 1207.

En base seize, on a besoin de 16 *chiffres :* 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *puis* a *(dix),* b *(onze),* c *(douze),* d *(treize),* e *(quatorze) et* f *(quinze).*

1207 | 16 → 7 | 75 ; 75 | 16 → 11 | 4 ; 4 | 16 → 4 | 0

Donc $1207 = 4b7_{16}$.

Savoir-faire Représenter en base dix un entier naturel donné en base k

Pour trouver la représentation en base dix d'un entier naturel donné en base k, on utilise le fait qu'en base k, le chiffre le plus à droite représente les unités, le précédent les paquets de k, le précédent les paquets de $k \times k = k^2$, le précédent les paquets de $k \times k \times k = k^3$, etc.

On retrouve les opérations réciproques de celles effectuées dans l'algorithme de conversion en base k : si $a_i \dots a_1 a_0$ est l'écriture de n en base k, alors

$$n = a_0 + a_1 k + a_2 k^2 + \dots + a_i k^i$$

On peut également suivre la méthode de Horner (vue en cours de mathématiques) pour faire ce calcul plus efficacement. Cela permet aussi plus naturellement de faire le calcul en lisant les chiffres de gauche à droite.

$$n = ((\dots(a_i k + a_{i-1})k + \dots)k + a_1)k + a_0$$

Exercice 2.8 avec corrigé Trouver la représentation en base dix du nombre 202413_5.

$$202413_5 = (3 \times 5^0) + (1 \times 5^1) + (4 \times 5^2) + (2 \times 5^3) + (0 \times 5^4) + (2 \times 5^5) = 6608$$

Exercice 2.9 avec corrigé Trouver la représentation en base dix du nombre $5ad9_{16}$.

Avec l'algorithme de Horner :

$$5ad9_{16} = ((5 \times 16 + 10) \times 16 + 13) \times 16 + 9 = 23257$$

2.1.3 La base deux

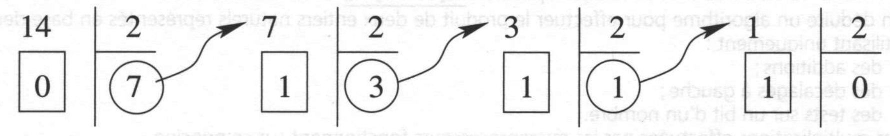
La mémoire des ordinateurs est constituée de circuits n'ayant chacun que deux états possibles : les ordinateurs comptent donc naturellement en base deux et chaque circuit représente un des deux chiffres de cette base : 0 ou 1.

Les nombres binaires sont plus difficiles à lire, mais le principe de la numération en base deux est en tout point similaire à celui de la numération dans les autres bases. Par exemple, le nombre treize s'écrit $\underline{1101}$: de droite à gauche, 1 unité, 0 deuzaine, 1 quatraine et 1 huitaine. L'écriture d'un entier naturel en binaire est en moyenne 3, 2 fois plus longue que son écriture en base dix, mais elle ne demande d'utiliser que deux chiffres. Le nombre $13 = \underline{1101}$ est donc représenté dans la mémoire d'un ordinateur par le mot 1101, c'est-à-dire par quatre circuits respectivement dans les états 1, 0, 1 et 1.

Dans la mémoire des ordinateurs, les circuits mémoire un bit sont souvent groupés par huit (les octets) et on utilise des nombres exprimés en notation binaire sur un, deux, quatre ou huit octets, soit 8, 16, 32 ou 64 bits. Cela permet de représenter les nombres de 0 à $\underline{1111\,1111} = 255$ sur un octet, de 0 à $\underline{1111\,1111\,1111\,1111} = 65\,535$ sur deux octets, de 0 à $\underline{1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111} = 4\,294\,967\,295$ sur quatre octets et de 0 à $\underline{1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111} = 18\,446\,744\,073\,709\,551\,615$ sur huit octets.

Lorsque l'on manipule des entiers naturels uniquement représentés de cette façon, dans certains langages on les appelle entiers *non signés*. En Python, toutefois, aucun type de données ne donne directement accès à ce genre de représentation : on ne peut manipuler que des entiers relatifs.

Exercice 2.10 avec corrigé Trouver la représentation en base deux du nombre 14.



Donc $14 = \underline{1110}$.

Exercice 2.11 avec corrigé Trouver la représentation en base dix du nombre $\underline{10101010}$.

$\underline{10101010} = (0 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) + (0 \times 2^4) + (1 \times 2^5) + (0 \times 2^6) + (1 \times 2^7) = 170$

Exercice 2.12 Trouver la représentation en base cinq des nombres 88, 627 et 1451.

Exercice 2.13 Trouver la représentation en base dix des nombres 4413_5 et 212_5.

Exercice 2.14 Trouver la représentation en base deux du nombre 10 000.

Exercice 2.15 Déterminer la taille du disque dur de votre machine en Mo. Trouver la représentation en base deux de ce nombre.

Exercice 2.16 Donner les représentations en base deux des nombres 1, 3, 7, 15, 31 et 63. Expliquer le résultat.

Exercice 2.17 Trouver la représentation en base seize des nombres suivants : 3718, 54920 et 482919.

Exercice 2.18 Trouver la représentation en base dix du nombre $\underline{1101\,1011}$.

Exercice 2.19 C'est en $\underline{111\,1011\,0001}$ qu'a été transmis le premier message sur Internet. Exprimer ce nombre en base dix. Le système tomba en panne après seulement $\underline{10}$ caractères transmis...

Exercice 2.20 Trouver la représentation en base dix des nombres $fedc_{16}$ et $6a5c1_{16}$.

Exercice 2.21 * Chercher sur le Web ce qu'est le système de numération Shadok. Est-ce un système de numération à position ? Si oui, en quelle base et avec quels chiffres ?

Exercice 2.22 Quelle est la représentation binaire du nombre 209 ? Et celle du nombre 46 ? Soit m un mot de 8 bits, n l'entier naturel représenté en binaire par le mot m, m' le mot obtenu en remplaçant dans m chaque 0 par un 1 et chaque 1 par un 0 et n' l'entier naturel représenté en binaire par le mot m'. Exprimer n et n' comme une somme de puissances de 2, montrer que $n + n' = 255$. Montrer que la représentation binaire du nombre $255 - n$ est obtenue en remplaçant dans celle de n chaque 0 par un 1 et chaque 1 par un 0.

Exercice 2.23 Montrer que les mots de n bits permettent de représenter tous les entiers de 0 à $2^n - 1$.

Inversement, combien de bits faut-il au minimum pour représenter tous les entiers dont l'écriture décimale comporte n chiffres ?

Exercice 2.24 On cherche à mettre au point une procédure de calcul du produit de deux entiers naturels représentés en binaire. Dans tout cet exercice, on pourra supposer par commodité que les entiers sont représentés sur 32 bits ; la généralisation à un nombre de bits quelconque ne pose cependant pas de difficulté particulière.

1 Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite, par exemple, $12 \times 10 = 120$. Quelle est l'opération équivalente pour les entiers naturels exprimés en base deux ? Illustrer cette remarque par des exemples.
On appelle cette opération *décalage à gauche*, car dans les microprocesseurs, au lieu d'ajouter un 0 à droite, on déplace tous les bits vers la gauche. Sur quels entiers cette opération n'est-elle pas possible sans commettre d'erreur ?

2 Quelle est l'opération effectuée sur un entier naturel en base deux si on lui fait subir deux décalages à gauche ? trois décalages ? n décalages ?

3 À l'aide des remarques précédentes, étant donnés deux entiers naturels n et m, exprimer le produit $n \times m$ en fonction de n et de la décomposition $\underline{m_i \dots m_1 m_0}$ de m en base deux.

4 En déduire un algorithme pour effectuer le produit de deux entiers naturels représentés en base deux utilisant uniquement :
- des additions ;
- des décalages à gauche ;
- des tests sur un bit d'un nombre.

Les multiplications effectuées par les microprocesseurs fonctionnent sur ce principe.

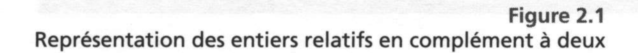
2.2 Représentation des entiers relatifs

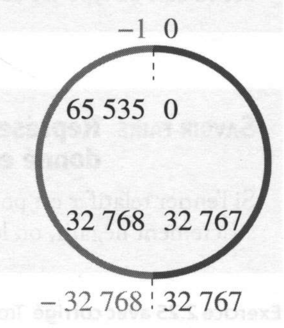
2.2.1 Notation en complément à deux

Il faut étendre aux entiers relatifs la représentation binaire des entiers naturels.

Une solution consiste à réserver un bit pour le signe de l'entier et à utiliser les autres pour représenter sa valeur absolue. Ainsi, avec des mots de 16 bits, en utilisant 1 bit pour le signe et 15 bits pour la valeur absolue, on pourrait représenter les entiers relatifs de $-\underline{111\,1111\,1111\,1111} = -(2^{15} - 1) = -32\,767$ à $\underline{111\,1111\,1111\,1111} = 2^{15} - 1 = 32\,767$. Cependant, cette méthode a plusieurs inconvénients, notamment l'existence de deux zéros, l'un positif et l'autre négatif.

On applique alors une autre méthode, qui consiste à représenter un entier relatif par un entier naturel. Si on utilise des mots de 16 bits, on peut représenter les entiers relatifs compris entre $-32\,768$ et $32\,767$: on représente un entier relatif x positif ou nul comme l'entier naturel x et un entier relatif x strictement négatif comme l'entier naturel $x + 2^{16} = x + 65\,536$, qui est compris entre 32 768 et 65 535. Ainsi, les entiers naturels de 0 à 32 767 correspondent aux entiers relatifs positifs ou nuls, à droite sur la figure 2.1 et les entiers naturels de 32 768 à 65 535 représentent les entiers relatifs strictement négatifs, à gauche sur cette même figure.


Figure 2.1
Représentation des entiers relatifs en complément à deux



Cette manière de représenter les entiers relatifs s'appelle la notation en complément à deux.

L'entier relatif -1 est représenté comme l'entier naturel 65 535, c'est-à-dire par le mot **1111 1111 1111 1111** (on ne le souligne pas car ici ce n'est pas tout à fait la représentation d'un nombre en base deux). On notera qu'il reste facile de déterminer le signe d'un entier représenté sous cette forme : un entier relatif positif ou nul est représenté par un entier naturel dont le premier bit vaut 0 ; *a contrario*, un entier relatif strictement négatif est représenté par un entier naturel dont le premier bit vaut 1.

Plus généralement, avec des mots de n bits, on peut représenter les entiers relatifs compris entre -2^{n-1} et $2^{n-1}-1$: on représente un entier relatif x positif ou nul comme l'entier naturel x (compris entre 0 et $2^{n-1}-1$) et un entier relatif x strictement négatif comme l'entier naturel $x+2^n$ (compris entre 2^{n-1} et 2^n-1).

ATTENTION La mémoire n'a pas de sens a priori.

Rien dans la représentation binaire n'indique comment il faut interpréter un mot mémoire donné. Le même mot peut donc être lu de plusieurs façons différentes, toutes aussi correctes les unes que les autres.

Il faut donc garder à part une trace des conventions de codage qu'on a décidé d'utiliser pour les différents mots mémoire que l'on manipule. De nos jours, cette gestion du sens de la mémoire n'est presque plus à la charge du programmeur : les langages de programmation évolués proposent pour la plupart un système de types qui permet d'associer facilement à chaque endroit de la mémoire une signification particulière.

EN PRATIQUE Dans les versions Python 2.x

Le type `int` désigne en principe des entiers relatifs représentés sur 32 bits (quatre octets), donc dont les valeurs vont de −2 147 483 648 à 2 147 483 647. Cependant, si l'on travaille sur une machine 64 bits, la représentation des entiers sera également faite sur 64 bits : on disposera donc des entiers de −9 223 372 036 854 775 808 à 9 223 372 036 854 775 807. L'ensemble des valeurs de type `int` dépend donc de la machine sur laquelle Python est exécuté !

SAVOIR-FAIRE Représenter en binaire sur n bits un entier relatif donné en décimal

Si l'entier relatif x est positif ou nul, on le représente comme l'entier naturel x. S'il est strictement négatif, on le représente comme l'entier naturel $x+2^n$.

Exercice 2.25 avec corrigé Trouver les représentations binaires sur huit bits des entiers relatifs 0 et -128.

L'entier relatif 0 *est représenté comme l'entier naturel* 0 *:* 00000000. *L'entier relatif* -128 *est représenté comme l'entier naturel* $-128+256=128$ *:* 10000000.

SAVOIR-FAIRE Représenter en décimal un entier relatif donné en binaire sur n bits

Si cet entier relatif est donné par le mot m, on commence par calculer l'entier naturel p représenté par ce mot. Si p est strictement inférieur à 2^n-1, c'est l'entier relatif représenté, s'il est supérieur ou égal à 2^n-1, l'entier relatif représenté est $p-2^n$.

Exercice 2.26 avec corrigé Trouver les représentations décimales des entiers relatifs dont les représentations binaires sur huit bits sont 00010111 et 10001100.

Le mot 00010111 *représente l'entier naturel* 23 *et donc l'entier relatif* 23. *Le mot* 1001100 *représente l'entier naturel* 140 *et donc l'entier relatif* $140 - 256 = -116$.

SAVOIR-FAIRE Calculer la représentation de l'opposé d'un entier relatif

On se place dans le cas où on connaît la représentation p sur huit bits de l'entier relatif x, et on cherche à calculer la représentation p' de son opposé.

Si l'entier relatif x est compris entre 0 et 127, alors il est représenté sur huit bits par l'entier naturel $p = x$ et son opposé $-x$ est représenté par l'entier naturel $p' = -x + 256 = 256 - p$. Si l'entier relatif x est compris entre -127 et -1, alors il est représenté par l'entier naturel $p = x + 256$ et son opposé $-x$ est représenté par l'entier naturel $p' = -x = 256 - p$. Donc, sauf quand $x = -128$ dont l'opposé n'est pas représentable, si un entier relatif x est représenté par l'entier naturel p, son opposé $-x$ est représenté par l'entier naturel $p' = 256 - p = (255 - p) + 1$.

Calculer $255 - p = \underline{11111111} - p$ est facile, puisqu'il suffit, dans la représentation binaire de p, de remplacer chaque 0 par un 1 et chaque 1 par un 0 (voir l'exercice 2.22). Il suffit ensuite d'ajouter 1 au nombre obtenu.

Exercice 2.27 avec corrigé Calculer la représentation sur huit bits de l'entier relatif 11, puis celle de son opposé.

L'entier relatif 11 *est représenté comme l'entier naturel* $11 = \underline{00001011}$.
Pour calculer la représentation de son opposé, on remplace les 0 *par des* 1 *et les* 1 *par des* 0, *ce qui donne* 1111 0100 *puis on ajoute* 1, *ce qui donne* 1111 0101. *On peut vérifier que ce nombre est bien la représentation de l'entier relatif* -11, *c'est-à-dire de l'entier naturel* $-11 + 256 = 245$.

2.2.2 Dépassements de capacité

Puisque les nombres représentables de cette façon sont limités, on peut se demander ce qui se passe lorsque l'on atteint ces limites. Dans l'exemple de calculs qui suit, on suppose une représentation des entiers sur 32 bits.

```
In [1]: a = 260

In [2]: a*a
Out[2]: 67600

In [3]: a*a*a
Out[3]: 17576000

In [4]: a*a*a*a
Out[4]: 4569760000
```

Le dernier résultat, `4569760000`, est représenté en binaire par 1 00010000 01100001 00000001 00000000. Si l'on reste dans une représentation sur 32 bits, le bit le plus à gauche est perdu : on ne mémorise que 00010000 01100001 00000001 00000000 = `274792704`, qui n'est évidemment pas le résultat attendu. On appelle ce phénomène *dépassement arithmétique* (*overflow* en anglais).

Il faut donc changer de représentation pour éviter de perdre la valeur du résultat. En Python, ce changement de représentation est fait automatiquement, mais c'est loin d'être le cas dans tous les langages de programmation. La plupart du temps, si le résultat d'un calcul dépasse les limites de la représentation des entiers, les bits surnuméraires sont purement et simplement perdus.

Dans ces langages, une conséquence inattendue de la notation en complément à 2 dans les dépassements est la suivante : le nombre 2 000 000 000 s'écrit en binaire 01110111 00110101 10010100 00000000 et donc l'opération 2 000 000 000 + 2 000 000 000 a pour résultat 11101110 01101011 00101000 00000000. Ce dernier nombre ne dépasse pas les 32 bits, mais dans la notation en complément à deux, il représente $-294\,967\,296$ et non pas 4 milliards. La somme de deux entiers positifs trop grands peut donc donner un résultat négatif ! De même, la somme de deux entiers négatifs peut résulter en un entier positif.

En Python, la seule limite pour la représentation des entiers, qu'ils soient naturels ou relatifs, est la mémoire disponible sur la machine. Les exemples précédents de dépassements arithmétiques ne se produisent donc pas.

EN PRATIQUE Dans les versions Python 2.x

Lorsque la capacité des entiers machine (32 ou 64 bits) a été dépassée, les nombres sont suivis du marqueur *L*, qui explicite qu'on passe dans un autre type appelé `long`. La dernière ligne de l'exemple précédent affiche donc plutôt

```
In [4]: a*a*a*a
Out[4]: 4569760000L
```

En Python 3.x, les types `int` et `long` sont fusionnés et, à toutes fins utiles, les entiers sont toujours de taille illimitée. Le marqueur *L* n'est plus utilisé.

Pour représenter des entiers de taille arbitraire, il a fallu prendre une certaine distance avec la représentation machine des entiers. Ainsi, dans l'implantation de référence de Python, sur les machines 32 bits, lorsque l'on veut représenter un entier x, on découpe la représentation binaire de x par paquets de 15 bits et on stocke les entiers correspondants dans un tableau d'entiers naturels sur 16 bits. Ce tableau contient donc les chiffres de x en base 2^{15}. De plus, à ce tableau on associe un entier `taille` qui donne le nombre de chiffres de x en base 2^{15} et dont le signe est le signe de x.

Par exemple, le nombre `4569760000` = 1 00010000 01100001 00000001 00000000 comporte 33 bits, que l'on découpe donc en 100 010000011000010 000000100000000. On a alors :

000000100000000 = 256

010000011000010 = 8386

100 = 4

Le nombre `4569760000` est donc représenté en mémoire par le tableau `[256, 8386, 4]`, auquel on associe le nombre `taille=3` pour signifier qu'il y a trois chiffres et que l'entier à représenter est positif. Le nombre `-4569760000` sera représenté par le même tableau associé au nombre `-3`. Enfin, cas particulier, le nombre `0` est représenté par un tableau vide associé au nombre de chiffres 0.

Sur une machine 64 bits, le principe reste le même, mais on découpe x en paquets de 30 bits et on le représente donc par le tableau de ses chiffres en base 2^{30}.

Exercice 2.28 Trouver la représentation binaire sur huit bits des entiers relatifs 127 et -127, puis de 101 et de -94.

Exercice 2.29 Quels entiers relatifs peut-on représenter avec des mots de 8 bits ? Combien sont-ils ? Mêmes questions avec des mots de 16 bits, 32 bits et 64 bits.

Exercice 2.30 Trouver la représentation décimale des entiers relatifs dont les représentations binaires sur 8 bits sont `01010101` et `10101010`.

Exercice 2.31 Montrer que le bit le plus à gauche vaut `1` pour les entiers relatifs strictement négatifs et `0` pour les entiers relatifs positifs ou nuls.

Exercice 2.32 On considère l'algorithme suivant, où les variables sont toutes des entiers relatifs représentés en complément à deux sur 32 bits.

$a \leftarrow x + y$
$a \leftarrow a - z$
$b \leftarrow x - z$
$b \leftarrow b + y$

1 Lesquelles de ces opérations risquent de donner lieu à un dépassement ?
2 Montrer que, quelles que soient les valeurs initiales de x et y, à la fin de l'algorithme les variables a et b contiennent la même valeur.
3 Que peut-on en conclure à propos de l'addition sur les entiers relatifs en machine ?

Exercice 2.33 * On considère des entiers relatifs sur 4 bits. Dessiner le cercle correspondant, sur le même modèle que celui de la figure 2.1, en plaçant les 16 nombres : $0, 1, 2, \ldots, 7, -1, -2, \ldots, -8$ à leur place. Relier les nombres opposés : 1 et -1, 2 et -2, etc. Quelle est l'interprétation géométrique de la fonction qui à chaque nombre associe son opposé ? Que penser du cas de -8 ?

Exercice 2.34 * Représenter les entiers relatifs 99 et 57 en binaire sur 8 bits. Ajouter les deux nombres binaires obtenus. Quel est l'entier relatif obtenu ? Pourquoi est-il négatif ?

Exercice 2.35 * Démontrer que l'addition de deux entiers relatifs en machine produit un dépassement arithmétique si et seulement si :

- ces entiers sont de même signe ;
- et le résultat obtenu en machine est de signe opposé au signe des opérandes.

On pourra commencer par supposer que les entiers sont représentés sur n bits, puis exprimer par un encadrement quels sont les entiers de chaque signe représentables dans ce format. Enfin, on en déduira un encadrement du résultat de la somme de deux entiers dans chacun des cas possibles.

De même, déterminer un critère permettant de détecter un dépassement arithmétique lors d'une soustraction de deux entiers relatifs.

Exercice 2.36 * *(À réaliser dans une version Python 2.x)*

À l'aide de la calculatrice Python, déterminez le plus grand entier représentable par le type `int` sur votre machine. Vérifiez si votre machine fonctionne en 32 bits ou en 64 bits et calculez la valeur théorique de ce plus grand entier pour vérifier votre réponse.

Exercice 2.37 * L'ensemble des entiers naturels machine écrits sur 16 bits, avec les dépassements de capacité expliqués dans ce chapitre (calcul sur un nombre de bits arbitraire puis troncature des bits surnuméraires) forme-t-il un groupe pour l'addition ? Forme-t-il un anneau pour l'addition et la multiplication ? Forme-t-il un corps ?

Exercice 2.38 ** L'ensemble des entiers relatifs machine écrits sur 16 bits, avec les dépassements de capacité expliqués dans ce chapitre forme-t-il un groupe pour l'addition ? Un anneau pour l'addition et la multiplication ? Un corps ?

Exercice 2.39 * L'ensemble des entiers longs de Python forme-t-il un groupe pour l'addition, en admettant que l'on dispose d'une mémoire illimitée ? Forme-t-il un anneau pour l'addition et la multiplication ? Forme-t-il un corps ?

Exercice 2.40 Un intérêt majeur de la notation des entiers relatifs en complément à deux est que les additions, les soustractions et les comparaisons peuvent toutes être effectuées au moyen d'une même opération, à quelques ajustements près.

Dans tout cet exercice, on supposera que les nombres sont représentés sur n bits.

1 Vérifier sur quelques exemples que la somme de deux entiers naturels représentés sur n bits peut se faire d'une façon similaire à celle que l'on apprend à l'école primaire pour les nombres en base 10 : calcul de la somme chiffre par chiffre, de droite à gauche, avec une retenue si nécessaire. Par exemple la somme $13 + 28$ s'écrit en binaire :

	32aines	seizaines	huitaines	quatraines	deuxaines	unités	
	1	1	1				*(retenues)*
			1	1	0	1	*(13)*
+		1	1	1	0	0	*(28)*
	1	0	1	0	0	1	*(41)*

2 En lien avec cette procédure d'addition, donner une caractérisation simple des cas où la somme de deux entiers naturels produit un dépassement de capacité.

3 Soient maintenant x et y deux entiers relatifs que l'on suppose représentables sur n bits. On appelle m et p les entiers naturels qui représentent respectivement x et y dans la notation en complément à 2. Démontrer que, quels que soient les signes de x et y, la somme binaire des entiers m et p effectuée comme à la question 1 donne bien l'entier naturel qui représente l'entier relatif $x + y$ en complément à 2, éventuellement à un dépassement arithmétique près.

4 On rappelle que pour déterminer la représentation de l'opposé d'un entier relatif, il suffit de partir de la représentation de cet entier, d'inverser tous ses bits puis d'ajouter 1 au résultat obtenu (voir exercice corrigé 2.27). En déduire une procédure simple pour calculer la différence de deux nombres relatifs en binaire en réutilisant encore une fois la procédure d'addition des entiers naturels. On pourra remarquer que l'étape « ajouter 1 au résultat obtenu » peut être intégrée dans l'addition au moyen d'une retenue.

5 Étant donnés deux entiers relatifs x et y, pour évaluer le test $x < y$, il suffit en théorie de calculer $x - y$ et d'observer le signe du résultat ; cependant, les dépassements de capacité faussent cette procédure puisqu'ils donnent à certains résultats des signes opposés à celui attendu. Il est cependant relativement facile de corriger ce problème. On appelle N le bit de signe du résultat apparent (calculé par la machine) de $x - y$: on aura donc $N = 1$ si ce résultat est négatif (voir exercice 2.31). On appelle V un bit égal à 1 si et seulement si le calcul de $x - y$ a donné lieu à un dépassement (voir exercice 2.35 pour la détermination de V). Déterminer une expression en fonction de N et de V qui vaut 1 si et seulement si $x < y$.

2.3 Représentation des nombres à virgule

2.3.1 L'arithmétique flottante

La notation binaire permet aussi de représenter des nombres à virgule. En notation décimale, les chiffres à gauche de la virgule représentent des unités, des dizaines, des centaines, etc. et ceux à droite de la virgule, des dixièmes, des centièmes, des millièmes, etc. De même, en binaire, les chiffres à droite de la virgule représentent des demis, des quarts, des huitièmes, des seizièmes, etc. Par exemple, le nombre un et un quart s'écrit $1,01$. Toutefois, cette manière de faire ne permet pas de représenter des nombres très grands ou très petits comme le nombre d'Avogadro ou la constante de Planck. On utilise donc une autre représentation similaire à la « notation scientifique » des calculatrices, sauf qu'elle est en base deux plutôt qu'en base dix et définie dans la norme IEEE 754. Un nombre est représenté sous la forme $sm2^n$ où s est le *signe* du nombre, n son *exposant* et m sa *mantisse*. Le signe est + ou -, l'exposant est un entier relatif et la mantisse est un nombre à virgule, compris entre 1 inclus et 2 exclu.

Quand on utilise 64 bits pour représenter un nombre à virgule, on utilise 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse.

- Le signe + est représenté par 0 et le signe $-$ par 1.
- L'exposant n est un entier relatif compris entre $-1\,022$ et $1\,023$; on le représente comme l'entier naturel $n + 1\,023$, qui est compris entre 1 et 2 046. Les deux entiers naturels 0 et 2 047 sont réservés pour des situations exceptionnelles ($+\infty$, $-\infty$, NaN, etc.), comme on va le voir ci-après.
- La mantisse m est un nombre binaire à virgule compris entre 1 inclus et 2 exclu, comprenant 52 chiffres après la virgule. Comme cette mantisse est comprise entre 1 et 2, elle a toujours le seul chiffre 1 avant la virgule ; il est donc inutile de le représenter et on utilise les 52 bits pour représenter les 52 chiffres après la virgule, ce qui donne une mantisse dont la précision réelle est de 53 bits.

Les nombres représentés sous cette forme sont appelés *nombres à virgule flottante*, puisque la virgule de la mantisse peut être « déplacée » par le biais de l'exposant.

SAVOIR-FAIRE Représenter en base dix un nombre à virgule flottante donné en binaire

On identifie le signe s, la mantisse m et l'exposant n. On interprète chacun comme un nombre décimal en n'oubliant pas de tenir compte du décalage de 1 023 pour l'exposant. On calcule enfin la quantité $sm2^n$.

Si l'on note s le bit de signe, $e_1 \dots e_{11}$ les bits d'exposant et $m_1 \dots m_{52}$ les bits de la mantisse, on peut également donner l'expression directe suivante du nombre représenté :

$$(-1)^s \times 2^{\underline{e_1 \dots e_{11}} - 1023} \times \left(1 + \sum_{i=1}^{52} m_i \frac{1}{2^i}\right)$$

Exercice 2.41 avec corrigé Trouver le nombre à virgule flottante représenté par le mot 1100010001101001001111000011100000000000000000000000000000000000.

Le signe est représenté par 1.

L'exposant est représenté par 10001000110.

La mantisse est représentée par 1001001111000011100000000000000000000000000000000000.

Le signe du nombre recherché est $-$. *Le nombre* $\underline{10001000110}$ *est égal à* 1 094 *et l'exposant est donc* $n = 1\,094 - 1\,023 = 71$. *La mantisse est*

$$\begin{aligned} m &= 1,1001001111000011100000000000000000000000000000000000 \\ &= 1 + 1/2 + 1/2^4 + 1/2^7 + 1/2^8 + 1/2^9 + 1/2^{10} + 1/2^{15} + 1/2^{16} + 1/2^{17} \\ &= (2^{17} + 2^{16} + 2^{13} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^2 + 2 + 1)/2^{17} \\ &= \frac{206\,727}{131\,072} \end{aligned}$$

Le nombre représenté est donc $-\frac{206\,727}{131\,072} \times 2^{71} \simeq -3,724\dots \times 10^{21}$.

2.3.2 Quelques cas particuliers

Il n'est pas indispensable de connaître toutes les exceptions à la règle présentée ci-avant, mais celle du zéro semble incontournable : si la mantisse est censée toujours commencer par un 1 implicite, il n'est en principe pas possible de représenter le zéro. Par convention, on décide qu'un nombre vaut zéro si et seulement si tous les bits de son exposant et de sa mantisse valent 0. Il reste un choix pour le bit de signe, il y a donc un zéro positif et un zéro négatif dans les nombres à virgule flottante.

Par extension, pour tout nombre dont l'exposant est composé de onze bits 0, on considère que le chiffre avant la virgule est un 0 implicite et que l'exposant vaut $-1\,022$. On dit ces nombres *dénormalisés* car ils ne respectent pas la convention habituelle sur le premier

chiffre de la mantisse. Ils permettent d'obtenir une répartition plus uniforme des valeurs infinitésimales représentables (voir exercice 2.51).

POUR ALLER PLUS LOIN Exceptions

Le codage prend également en compte des valeurs exceptionnelles : $+\infty$, $-\infty$ et enfin les *NaN* (*Not a Number*) qui signalent en général une erreur de calcul, par exemple une division par zéro ou la racine carrée d'un nombre négatif. Ces valeurs non numériques sont représentées respectivement par les mots de 64 bits suivants :

0 11111111111 0000000000000000000000000000000000000000000000000000

1 11111111111 0000000000000000000000000000000000000000000000000000

0 ou 1 puis 11111111111 puis tout mot de 52 bits non tous nuls.

2.3.3 Dépassements de capacité et problèmes de précision

De même que les entiers, les nombres à virgule flottante possèdent certaines limites inévitables. En voici un rapide inventaire, qui sera détaillé dans la troisième partie *Ingénierie numérique et simulation*, au cours de laquelle ces problèmes se poseront concrètement.

Les nombres à virgule flottante étant représentés sur un nombre donné de bits, il existe forcément un nombre maximal représentable dans ce format. Plus précisément, 64 bits ne suffisent plus si la représentation du nombre demande :

- un exposant supérieur à 1 023, qui est le plus grand représentable sur 11 bits ;
- ou un exposant égal à 1 023 et une mantisse supérieure à la plus grande mantisse représentable sur 52 bits, c'est-à-dire 1 (implicite) suivi de 52 fois le chiffre 1 après la virgule.

Tout calcul dont le résultat dépasse cette limite produit une situation qui est également appelée *dépassement arithmétique* ; cependant, au lieu de produire une valeur par simple troncature des bits surnuméraires, on utilise les nombres spéciaux $+\infty$ et $-\infty$, selon le signe du résultat du calcul. Dans les opérations qui suivent, ceux-ci respectent les règles de calcul usuelles sur les limites et produisent un *NaN* lorsqu'aucune règle de calcul ne peut être appliquée sans ambiguïté.

Une situation similaire mais qui n'existait pas pour les entiers se produit lorsque l'on veut représenter un nombre trop proche de 0 :

- Son exposant est inférieur à $-1\,022$, le plus petit exposant représentable sur 11 bits.
- Ou ce nombre est inférieur en valeur absolue au plus petit nombre dénormalisé.

On parle alors de *dépassement par valeurs inférieures* ou de *soupassement arithmétique* (en anglais *underflow*). Selon les cas, le résultat d'un calcul qui tombe dans cette plage de valeurs peut soit être arrondi à zéro (le signe du résultat est cependant conservé), soit produire une erreur.

2.3.4 Les arrondis

Il est rare que le résultat d'un calcul faisant intervenir deux nombres à virgule flottante donne un résultat représentable exactement sur 64 bits. Même sans effectuer de calcul, les nombres décimaux ne sont pour la plupart pas représentables exactement dans ce format. Par exemple, le nombre 0, 4 admet pour développement en base 2 :

$$\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^6}+\frac{1}{2^7}+\frac{1}{2^{10}}+\frac{1}{2^{11}}+\frac{1}{2^{14}}+\frac{1}{2^{15}}+\ldots=\underline{0,011001100110011\ldots}$$

qui est un développement infini périodique.

Si l'on essaye de construire sa représentation sur 64 bits, on obtient :

signe	exposant	mantisse	bits non représentés
0	01111111101	1001100110011001... 1001100110011001	10011001...

La représentation en virgule flottante sera donc forcément une valeur approchée de ce nombre. Par défaut, la norme IEEE 754 impose que les nombres à virgule soient arrondis à la valeur représentable la plus proche. Ici, la valeur des bits non représentés est supérieure à la moitié de la valeur du dernier bit de la mantisse, donc on arrondit la mantisse vers le haut. On notera que cette opération est facile à effectuer, puisqu'il suffit d'arrondir vers le haut si le premier bit non représenté vaut 1, vers le bas sinon. Seul cas particulier, si la valeur des bits non représentés est *exactement* égale à la moitié de la valeur du dernier bit de la mantisse, alors le nombre est arrondi de telle sorte que son dernier bit vaille 0.

La valeur approchée choisie pour 0, 4 est donc la suivante :

$$\begin{aligned}&0,0110011001100110011001100110011001100110011001100110011010\\&=\frac{7\,205\,759\,403\,792\,794}{18\,014\,398\,509\,481\,984}\\&\simeq 0,400000000000000022204460492503\end{aligned}$$

ATTENTION Représentation des décimaux en binaire

À cause de la base utilisée, il est impossible de représenter exactement la plupart des nombres décimaux, plus précisément tous ceux qui ne s'écrivent pas sous la forme $\frac{k}{2^n}$.

Des nombres qui habituellement ne posent pas de problème dans les calculs en mathématiques deviennent ainsi une source d'erreurs multiples.

D'autres erreurs d'arrondis se présentent lorsque l'on effectue des calculs, notamment entre des nombres dont les ordres de grandeur sont très différents. Dans ce cas, on effectue le calcul comme si la précision était infinie, puis on arrondit le résultat obtenu.

$$1 + 2^{-54} = 1,\underline{00000000000000000000000000000000000000000000000000001}$$

demanderait 54 bits de mantisse pour être représenté exactement ; il est donc arrondi à 1. Cet arrondi peut paraître bénin, mais s'il se produit dans une suite de calculs, il peut contrevenir aux propriétés habituelles des opérations sur les nombres réels. Par exemple, en Python, on peut observer

```
In [5]: 1 + 2**-54 - 1

Out[5]: 0.0
```

La somme est en effet calculée en premier et arrondie à 1, puis la différence est calculée et le résultat vaut donc 0. En revanche

```
In [6]: 1 - 1 + 2**-54

Out[6]: 5.551115123125783e-17
```

Comme la différence est calculée en premier, le résultat est ici celui attendu ; mais on voit que l'addition sur les nombres à virgule flottante n'est pas associative, puisque $(1+(-1)) + 2^{-54} \neq 1 + ((-1) + 2^{-54})$. Des erreurs d'arrondi similaires peuvent se produire avec toutes les opérations usuelles ; on pourra pour s'en convaincre étudier l'évaluation en machine de $\left(1 + 2^{-50}\right)^2$.

On retiendra qu'il n'est pas possible de savoir de façon certaine si le résultat d'un calcul est égal à sa valeur théorique. La principale conséquence en est qu'un test du type `a==b` n'a en général pas de sens si `a` et `b` sont deux nombres à virgule flottante, puisque ceux-ci ont pu subir des erreurs d'arrondis. On remplacera donc un tel test par une condition de la forme `abs(a-b)<epsilon` où `epsilon` est une valeur proche de zéro, choisie en fonction du problème à traiter et de l'ordre de grandeur des erreurs auxquelles on peut s'attendre sur `a` et `b`.

Exercice 2.42 Trouver le nombre à virgule flottante représenté par le mot binaire 0001000000111101001110010101100000000000000000000000000000000000.

Exercice 2.43 Comment est représenté le nombre $2^{-1\,022}$ (qui est égal à environ $2,225 \times 10^{-308}$) ?

Exercice 2.44 Représenter les nombres suivants avec la valeur la plus précise que vous pourrez trouver dans la littérature : la vitesse de la lumière dans le vide, le nombre d'Avogadro, la constante de Coulomb. Pour lesquels de ces nombres la précision des nombres à virgule flottante sur 64 bits est-elle insuffisante ? Connaissez-vous une grandeur physique ou chimique pour laquelle l'étendue des nombres à virgule flottante est insuffisante ?

Exercice 2.45 Par quel mot binaire est représenté en machine le nombre entier 13 ? Et le nombre à virgule flottante 13, 0 ?

Exercice 2.46 À combien de décimales environ correspondent 52 chiffres binaires après la virgule ?

Exercice 2.47

1 Quel est le plus grand nombre que l'on peut représenter en virgule flottante sur 64 bits ?
2 Quel est le plus petit nombre, donc négatif, que l'on peut représenter en virgule flottante sur 64 bits ?
3 Quel est le plus petit nombre normalisé strictement positif que l'on peut représenter en virgule flottante sur 64 bits ?

Il est à noter qu'en Python, la commande `float_info` de la bibliothèque `sys` fournit ces informations, ainsi que la plupart des caractéristiques de la représentation des nombres en virgule flottante : taille de la mantisse, différence relative entre deux valeurs consécutives, etc. (Attention, les valeurs possibles pour l'exposant sont décalées d'une unité.)

Exercice 2.48 Reprendre les questions de l'exercice précédent pour les nombres à virgule flottante *simple précision* : ceux-ci sont représentés sur 32 bits, avec 1 bit de signe, 8 bits d'exposant et 23 bits de mantisse. Ils sont utilisés dans certains langages de programmation, notamment pour des raisons d'économie de mémoire et de rapidité des calculs ; en Python cependant, leur utilisation n'apporterait pas un bénéfice significatif.

Exercice 2.49 Déterminer l'écriture binaire et une valeur décimale approchée du plus grand nombre à virgule flottante sur 64 bits strictement inférieur à 1.
Même question avec le plus petit nombre strictement supérieur à 1.

Exercice 2.50 * En s'inspirant de l'algorithme de conversion des entiers naturels en base k, concevoir un algorithme de conversion d'un nombre décimal vers son écriture en virgule flottante.

On pourra commencer par le cas des nombres compris entre 1 et 2, pour lesquels l'exposant est connu.

On traitera ensuite les nombres inférieurs à 1 ou supérieurs à 2 en se ramenant au cas précédent.
Enfin, pour que l'algorithme soit complet, on n'oubliera pas le signe.

Exercice 2.51 * Cet exercice a pour objectif d'étudier l'intérêt des nombres dénormalisés et quelques-unes de leurs particularités.

1 Soit x_1 le plus petit nombre à virgule flottante normalisé strictement positif représentable sur 64 bits, et x_2 le plus petit nombre représentable strictement supérieur à x_1.
Calculer x_1 et x_2. Combien vaut l'écart relatif $\dfrac{x_2 - x_1}{x_1 - 0}$?
2 Refaire la même étude avec les deux plus petits nombres dénormalisés y_1 et y_2.
3 Calculer également l'écart relatif $\dfrac{y_2 - y_1}{x_2 - x_1}$ et interpréter ce résultat en termes de densité des nombres représentables dans différents intervalles.
4 Expliquer pourquoi l'exposant des nombres dénormalisés doit être $-1\,022$ et non pas $-1\,023$ comme serait interprété l'exposant 00000000000 dans le cas d'un nombre normalisé.

Exercice 2.52 Quelle précision perd-on si on divise par deux un nombre à virgule flottante sur 64 bits avant de le remultiplier par deux ?

Exercice 2.53 Montrer que, pour comparer deux nombres à virgule flottante de même signe, il suffit de les comparer bit par bit : si leurs bits les plus à gauche sont différents, celui dont ce bit vaut 1 est le plus grand ; sinon on compare leurs bits suivants jusqu'à trouver deux bits différents pour pouvoir appliquer cette règle.
(Cette règle ne s'applique bien entendu pas aux infinis et aux NaN*, dont on rappelle que l'exposant vaut* 11111111111.*)*

Exercice 2.54 *

1 Montrer qu'à chaque multiplication de deux nombres à virgule flottante, comme on arrondit le calcul en ne gardant que 52 chiffres après la virgule, on introduit une erreur relative de l'ordre de 2^{-52}.
2 Quelle est la valeur de cette erreur en base dix ?
3 Si on fait plusieurs multiplications, ces erreurs s'accumulent. Quelle est l'erreur relative d'un calcul qui est formé d'un million de multiplications, qui dure quelques millisecondes sur un ordinateur usuel ?

Exercice 2.55 On considère le programme Python suivant :

```
x = 1.0
y = x + 1.0
while y - x == 1.0:
    x = x * 2.0
    y = x + 1.0
```

1 Si l'on calculait sur des nombres décimaux exacts, que se passerait-il lors de l'exécution de ce programme ?
2 Écrire ce programme dans un éditeur Python et l'exécuter. Que constate-t-on ?
3 Modifier le programme de façon à déterminer au bout de combien d'exécutions du corps de la boucle il s'arrête, ainsi que la valeur de x à la fin de cette exécution.
4 Comment est représentée cette dernière valeur de x ? Et celle de y ?
5 Proposer une explication de ce comportement.

Exercice 2.56 On considère le programme Python suivant :

```
a = 0.0
for n in range(10):
    a = a + 0.1
    print(repr(a))
```

1 Si l'on calculait sur des nombres décimaux exacts, que se passerait-il lors de l'exécution de ce programme ?
2 Écrire ce programme et l'exécuter. Que constate-t-on ?
3 Vérifier que la représentation binaire de $0,1$ est
0011111110111001100110011001100110011001100110011001100110011010.
Quel nombre décimal cette représentation désigne-t-elle en réalité ?
4 En déduire les représentations binaires des différentes valeurs prises par `a` au cours de l'exécution de ce programme et les nombres décimaux que cette représentation désigne en réalité.
5 Expliquer l'affichage obtenu.

Exercice 2.57 * La représentation binaire en virgule flottante sur 64 bits permet-elle de représenter :

1 tous les entiers relatifs représentables sur 64 bits ?
2 plus de nombres distincts que la représentation des entiers relatifs sur 64 bits ? ou moins ?
3 des nombres décimaux qui ne soient pas entiers ?
4 des nombres rationnels qui ne soient pas décimaux ?
5 des nombres réels qui ne soient pas rationnels ?

Exercice 2.58 * Donner des exemples de nombres :

1 entiers ;
2 décimaux non entiers ;

qui ne sont pas représentables exactement selon la norme IEEE 754.

Dans les deux cas, on proposera des nombres impossibles à représenter exactement :

1 pour des raisons de précision ;
2 parce qu'ils sont trop grands, trop petits ou trop proches de 0.

Exercice 2.59 * L'ensemble des nombres à virgule flottante sur 64 bits forme-t-il un groupe pour l'addition ? Un anneau pour l'addition et la multiplication ? Un corps ?

On rappelle que les opérations sont faites comme si la précision était infinie, puis que le résultat obtenu est arrondi au plus proche nombre à virgule flottante représentable.

Deuxième partie

Algorithmique et programmation

Dans cette partie, nous présentons les notions clés de l'algorithmique (chapitres 3 et 4) en nous attachant systématiquement à démontrer que les algorithmes écrits produisent le résultat attendu. Nous abordons également la traduction de ces algorithmes sous forme de programmes. Nous présentons ensuite la notion de fonction (chapitre 5) qui permet d'organiser les programmes et leur développement, ainsi que les fonctions récursives, qui font partie du programme de deuxième année. Nous montrons enfin comment évaluer l'efficacité d'un algorithme, et nous présentons une première structure de données : les tableaux (chapitre 6).

3

Expressions : types et opérations

Dans ce chapitre, nous voyons comment manipuler des valeurs, c'est-à-dire d'une part comment les exprimer*, qu'il s'agisse de constantes ou de valeurs obtenues par calculs, et d'autre part comment les* stocker *à l'aide de la notion de variables. Nous utilisons l'interpréteur interactif Python pour illustrer ces notions.*

3.1 Expressions et types simples

3.1.1 Expression

Une expression est une suite de caractères définissant une valeur. Pour calculer cette valeur, la machine doit évaluer l'expression. Voici des exemples d'expressions : `42`, `1+4`, `1.2 / 3.0`, `x+3`.

Le résultat du calcul peut dépendre de l'environnement au moment où le calcul est effectué. Ainsi une expression est-elle plus complexe à évaluer qu'une valeur constante. On parlera *des* valeurs possibles d'une expression.

En Python, pour évaluer une expression, il suffit de la saisir dans un interpréteur, qui calcule et affiche alors la valeur qu'il a calculée :

```
In [1]: 42
Out[1]: 42
```

```
In [2]: 1+4
Out[2]: 5
```

Les *valeurs* en Python sont *typées* selon l'objet qu'elles représentent. Une valeur peut ainsi être de type entier, de type flottant, de type *chaîne de caractères*... Des types similaires existent dans la plupart des langages de programmation. Leur représentation en mémoire varie beaucoup d'un langage à l'autre, mais ce sont souvent les mêmes objets que l'on cherche à traduire.

Une *expression* en Python n'a pas de type *a priori* car le type de sa valeur dépend de l'environnement, plus précisément des types de ses sous-expressions. Pour simplifier, on considèrera dans un premier temps des expressions dont les valeurs sont toutes d'un même type. L'expression `42` est de type entier alors que `1.2 / 3.0` est de type flottant.

Pour afficher le type d'une expression après l'avoir évaluée, on utilise **`type`** :

```
In [3]: type(42)
Out[3]: <class 'int'>
```

```
In [4]: type(1.2 / 3.0)
Out[4]: <class 'float'>
```

Le mot qui suit **`class`** indique le type de la valeur, entier (*int* en anglais) pour la première expression et flottant (*float* en anglais) pour la seconde.

On ignorera ici ce mot **`class`** qui fait référence au fait que Python est un langage orienté objet.

Dans la plupart des langages de programmation, une expression est :

- soit une constante, comme `42` ;
- soit un nom de variable (valeurs littérales comme *x* ou *compteur*) ;
- soit une expression entre parenthèses, comme en mathématiques `(2-3)` ;
- soit composée de plusieurs expressions réunies à l'aide d'un opérateur, comme `1 + (3 * 4)` où les expressions `1` et `3 * 4` sont réunies par l'opérateur `+` ;
- soit composée d'une fonction appliquée à d'autres expressions, comme **`fact`**`(4)`.

Dans les prochaines sections, on présente les constantes et les opérateurs sur les types simples usuels, puis on verra la notion de variable. On présentera enfin des types plus complexes. La notion de fonction sera l'objet du chapitre 5.

3.1.2 Entiers

Constantes

Les constantes entières sont écrites en base 10 avec des nombres ayant autant de chiffres qu'on le souhaite. On dit que Python utilise des entiers longs ; par abus de langage, on parle également de précision arbitraire pour exprimer que tous les chiffres significatifs sont mémorisés.

Pour écrire un nombre négatif, on utilise - devant le nombre.

Par exemple :

```
In [5]: 42
Out[5]: 42
```

```
In [6]: -12
Out[6]: -12
```

```
In [7]: 12345678910111213141516171819
Out[7]: 12345678910111213141516171819
```

Opérateurs sur les entiers

Les opérateurs sur les entiers sont :

- **l'addition** utilisant le symbole `+` ;

```
In [8]: 2+4
Out[8]: 6
```

- **la soustraction** utilisant le symbole `-` ;

```
In [9]: 2-4
Out[9]: -2
```

- **la multiplication** utilisant le symbole `*` ;

```
In [10]: 2*4
Out[10]: 8
```

- **le quotient et le reste dans la division euclidienne** utilisant respectivement les symboles `//` et `%`.

```
In [11]: 17 // 5
Out[11]: 3
```

```
In [12]: 17 % 5
Out[12]: 2
```

  Dans le contexte de la programmation on parle de *division entière* et de *modulo* plutôt que de quotient et de reste. Ainsi l'expression `2 % 5` se lit *2 modulo 5* ;

- **l'exponentiation** utilisant le symbole `**` (`n ** m` calcule n^m). En combinant cela avec la précision arbitraire, on peut déjà faire des calculs élaborés :

```
In [13]: 2 ** 100
Out[13]: 1267650600228229401496703205376
```

- **l'opposé** utilisant également le symbole `-` mais devant une expression, on parle alors de position *préfixe*.

```
In [14]: -(2+3)
Out[14]: -5
```

POUR ALLER PLUS LOIN La division euclidienne appliquée aux nombres négatifs

On rappelle ici une variante d'un théorème vu en mathématiques :

Théorème. Pour tous entiers n, m avec $m \neq 0$, il existe un unique couple $(q, r) \in \mathbb{Z}^2$ tel que $n = qm + r$ avec r compris entre 0 inclus et m exclu.

q et r sont alors appelés le quotient et le reste dans la division euclidienne de n par m. Le reste r est éventuellement négatif (si m l'est).

Par exemple, $8 = 1 \times 5 + 3$ donc

```
In [15]: 8 // 5
Out[15]: 1
```

et

```
In [16]: 8 % 5
Out[16]: 3
```

Mais on a également $1 = 0 \times 2 + 1$ donc

```
In [17]: 1 // 2
Out[17]: 0
```

Attention, ce théorème est différent de celui que l'on utilise usuellement dans le cours de mathématiques :

Théorème. Pour tous entiers n, m avec $m \neq 0$, il existe un unique couple $(q, r) \in \mathbb{Z} \times \mathbb{N}$ tel que $n = qm + r$ et $0 \leqslant r < |m|$.

Cette différence s'exprime lorsque l'on divise par des nombres négatifs.

Si on divise 3 par -2 on obtient ces deux calculs :

$$3 = -2 \times -2 + (-1) = -1 \times -2 + 1$$

En Python, c'est le premier couple $(-2, -1)$ qui est renvoyé car $-2 < -1 \leqslant 0$, mais en mathématiques, on considère le second couple $(-1, 1)$ car $0 \leqslant 1 < |-2| = 2$.

La division euclidienne ne pouvant s'effectuer qu'avec un diviseur non nul, il est possible qu'une expression n'ait pas de valeur car elle reviendrait à diviser par 0. Python l'indique par un message d'erreur :

```
In [18]: 1 // 0

ZeroDivisionError: integer division or modulo by zero
```

Précédence des opérateurs sur les entiers

Si on considère l'expression `2+3*4`, il y a deux manières de l'évaluer : `(2+3)*4` ou `2+(3*4)`. Pour ne pas avoir à demander à l'utilisateur laquelle des deux expressions calculer, les langages de programmation appliquent automatiquement des règles de placement des parenthèses. Celles-ci généralisent les règles de priorité des opérations usuelles ; on parle de *précédence*.

Dans une expression, en l'absence de parenthèses, on calcule en priorité :

- les exponentiations (opérateur `**`) ;
- puis les multiplications, divisions entières et modulos (opérateurs `*`, `//` et `%`) ;
- et enfin les additions et soustractions (opérateurs `+` et `-`).

Ces règles ne résolvent pas toutes les ambiguïtés. Il reste toujours le cas des expressions telles que `1-3-2`, qu'on peut comprendre comme `(1-3)-2` (de valeur `-4`) ou comme `1-(3-2)` (de valeur `0`). Pour la plupart des opérateurs, on choisit toujours la première possibilité (associativité à gauche). Une exception notable est l'exponentiation, qui dans une expression comme `2**1**3` est calculée à droite d'abord.

Exercice 3.1 Écrire une expression qui permet de déterminer le dernier chiffre de $2\,013^{2\,013}$ sans afficher ce nombre en entier.

Exercice 3.2 Écrire des expressions formées uniquement :

- des opérateurs sur les entiers,
- de parenthèses,
- et du chiffre 5 utilisé au maximum cinq fois.

Chercher à obtenir un maximum de valeurs entières positives distinctes. On s'efforcera d'écrire des expressions comportant le moins de parenthèses possibles.

1 Quel est le plus petit entier que l'on ne peut pas obtenir avec une telle expression ?
2 Quel est le plus grand entier que l'on peut obtenir ?

3.1.3 Flottants

Constantes

Les nombres à virgule flottante, souvent appelés plus simplement flottants, ont été vus au chapitre 2.

Pour écrire un nombre flottant (sous réserve qu'il ne dépasse pas la capacité de ce type), on utilise le symbole . à la place de la virgule.

```
In [19]: 3.2
Out[19]: 3.2
```

```
In [20]: -0.01
Out[20]: -0.01
```

EN PRATIQUE Écriture abrégée des flottants

Lorsque l'un des deux côtés du symbole . est 0 on peut l'omettre :

```
In [21]: 2.
Out[21]: 2.0
```

```
In [22]: .3
Out[22]: 0.3
```

En toute généralité, on peut vouloir écrire un flottant sous la forme $m10^n$ où m la mantisse et n l'exposant sont des nombres relatifs. Par exemple, $0,0012 = 12 \times 10^{-4}$ et $-1\,230 = -123 \times 10^1$. Pour traduire cela en Python, on utilise la lettre e pour séparer la mantisse et l'exposant :

```
In [23]: 12e-4
Out[23]: 0.0012
```

```
In [24]: -123e1
Out[24]: -1230.0
```

On remarque que Python ajoute .0 même quand il s'agit d'un entier au sens mathématique du terme. Cela permet de rappeler d'un coup d'œil qu'il est du type flottant.

Opérateurs sur les flottants

Les opérateurs +, -, *, ** sont également définis sur les flottants, avec les mêmes règles de précédence.

```
In [25]: 1.2+3.1
Out[25]: 4.3
```

Ici on remarque que les flottants offrent une précision moins bonne que les entiers pour les calculs sur des valeurs entières :

```
In [26]: 2.0 ** 100.0
Out[26]: 1.2676506002282294e+30
```

Lors d'une opération entre un flottant et un entier, l'entier considéré est converti à la volée en flottant.

```
In [27]: 1.2 * 2
Out[27]: 2.4
```

Les flottants disposent également d'un opérateur de division flottante /.

```
In [28]: 1.0 / 3.0
Out[28]: 0.3333333333333333
```

EN PRATIQUE Dans les versions Python 2.x

En Python 2.x, l'opérateur de division / équivaut sur les entiers à la division entière :

```
In [29]: 1 / 3
Out[29]: 0
```

Pour obtenir le quotient décimal de deux entiers, il suffit d'indiquer que l'un des arguments doit être pris comme un flottant, soit en ajoutant un ., soit en utilisant le convertisseur `float` :

```
In [30]: 1. / 3
Out[30]: 0.3333333333333333
```

```
In [31]: float(1) / 3
Out[31]: 0.3333333333333333
```

Dans les versions 3.x de Python, l'opérateur / n'existe que pour les flottants; quand on l'applique à des entiers, ces derniers sont convertis en flottants et le résultat est un flottant égal à leur quotient décimal :

```
In [32]: 1 / 3
Out[32]: 0.3333333333333333
```

SAVOIR-FAIRE Choisir un type approprié pour représenter un nombre

En Python, un nombre est représenté soit par un entier, soit par un flottant.

- Lorsque l'exactitude du résultat a de l'importance, on choisit des calculs en entiers (de précision arbitraire), puisque les flottants n'apportent qu'une précision de l'ordre d'une quinzaine de chiffres décimaux.

- Pour représenter une grandeur physique, on choisit plutôt des flottants. Même lorsqu'il est possible d'utiliser des entiers pour représenter des grandeurs physiques, cela n'a généralement que peu d'intérêt, celles-ci étant pour la plupart connues avec seulement une dizaine de chiffres significatifs.
- Enfin, si on doit effectuer des calculs sur des données dont les ordres de grandeur sont très différents, les flottants ne sont pas appropriés (voir section 2.3.4).

Exercice 3.3 avec corrigé Quel est le type approprié lorsque l'on veut :

1 représenter la taille d'un individu en mètres ;
2 représenter le nombre d'Avogadro ;
3 calculer le plus petit multiple de 7 supérieur à 10^{100} ;
4 calculer la force gravitationnelle exercée sur la Terre par l'ensemble de ses satellites.

1 *Un flottant : indiquer qu'une personne mesure environ $2\,m$, à une précision d'un mètre près est pour le moins imprécis !*
2 *Le nombre d'Avogadro pourrait certes être représenté sous forme d'un entier à 24 chiffres, mais il n'est connu qu'à $2,7 \times 10^{16}$ près. Il est donc plus raisonnable de dire que ce nombre est d'environ $6,022\ 141\ 29 \times 10^{23}$ (à $\pm 0,000\ 000\ 27 \times 10^{23}$ près). L'incertitude relative étant de 5×10^{-8}, il est possible de le représenter sans difficulté par un flottant.*
3 *Des entiers : il faut une précision de l'ordre de 100 chiffres et on ne peut pas se permettre une erreur sur le chiffre des unités.*
4 *L'attraction exercée par la Lune sur la Terre est très nettement supérieure à celle exercée par les satellites artificiels (de l'ordre de 10^{17} fois supérieure pour de nombreux satellites géostationnaires par exemple). Un calcul en flottants introduirait donc rapidement des erreurs d'arrondis.*
On peut tout de même se demander s'il est bien raisonnable de vouloir effectuer un tel calcul puisque, dans le résultat final, les chiffres significatifs proviendront uniquement de l'attraction lunaire.

POUR ALLER PLUS LOIN La représentation des grandeurs monétaires

La représentation des quantités monétaires en comptabilité est un cas intéressant : d'une part, on a besoin de montants inférieurs à une unité (on veut représenter des centièmes d'euros voire moins : les cours des devises sont en général exprimés en euros à 10^{-4} près) et d'autre part le calcul en virgule flottante n'est pas adapté en raison des erreurs liées aux approximations (voir exercice 3.5). Pour un comptable, il n'est pas acceptable qu'un centime d'euro manque à l'appel !

Il est donc préférable de représenter les quantités monétaires par des entiers. On prendra par exemple comme unité le centime d'euros, voire 10^{-4} euro. Il faut évidemment prêter attention à la question des divisions : lorsqu'un comptable répartit un montant à payer en vingt-trois parts, il est nécessaire pour lui que la somme des vingt-trois parts soit rigoureusement égale au montant à payer ; il convient donc de voir comment répartir le reste de la division euclidienne.

Certains langages de programmation proposent cependant un type de données prévu pour représenter les décimaux. Python possède un module implantant un tel type de données : le module `decimal`.

Savoir-faire Convertir une expression numérique d'un type à un autre

Pour utiliser le nombre 42, on écrira `42` si l'on veut utiliser un entier et `42.0` ou simplement `42.` si l'on veut un flottant.

On peut, en cas de besoin, convertir un entier en flottant en lui appliquant la fonction **`float`** et un flottant en un entier par la fonction **`int`**. Attention : **`int`**`(x)` ne calcule pas la partie entière de x mais le tronque, c'est-à-dire calcule la partie entière de $|x|$, puis affecte le résultat du signe de x.

Enfin, si on utilise un entier dans une opération qui requiert un flottant (quotient décimal ou racine carrée par exemple), Python effectue la conversion automatiquement. Il est tout de même recommandé d'écrire explicitement les conversions là où on souhaite qu'elles aient lieu, ne serait-ce que dans un souci de lisibilité.

Exercice 3.4 avec corrigé Les expressions `8.5/2.5`, **`int`**`(8.5)/`**`int`**`(2.5)` et **`int`**`(8.5/2.5)` sont-elles équivalentes ?

Non et à deux titres. Au niveau des types d'abord, la dernière expression est un entier, alors que les deux premières sont des flottants puisqu'elles sont le résultat de quotients décimaux /.
Ensuite, si on calcule leurs valeurs :
- *Dans la première expression, le quotient* `8.5/2.5` *s'évalue à* `3.4`.
- *Dans la seconde expression,* **`int`**`(8.5)` *et* **`int`**`(2.5)` *s'évaluent respectivement à* `8` *et* `2`. *Ces valeurs sont ensuite automatiquement reconverties en flottants pour effectuer la division, mais leur partie décimale a été perdue au passage :* `8.0/2.0` *s'évalue à* `4.0`.
- *Dans la troisième expression, on retrouve le quotient* `8.5/2.5` *de la première expression. Sa valeur* `3.4` *est ensuite tronquée pour obtenir l'entier* `3`.

On remarquera que, même en effectuant une conversion supplémentaire vers les entiers, les deux dernières expressions ne peuvent être rendues égales.

Exercice 3.5 *

1 Qu'affiche Python lorsqu'on lui demande de calculer `1 - 1./3 - 1./3 - 1./3` ? Expliquer.
2 Et quand on lui demande `1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2` ? Expliquer.
3 Quel est l'effet de la fonction **`round`** ? Que renvoie Python lorsqu'on lui demande de calculer **`round`**`(1.05,1)`, **`round`**`(10.05,1)`, **`round`**`(100.05,1)` ? Expliquer.

Exercice 3.6 Quelle est la valeur des expressions suivantes, lorsqu'elle existe ?

- `1.5 + 1.5`
- `1.5 - 1`
- `3. * 7`
- `(1-2).`
- `.5-.3`
- `4 / (9 - 3**2)`
- **`float`**`(7 // 2)`

Vérifier les réponses dans un interpréteur.

Exercice 3.7 Les expressions **`int`**`(a/b)` et `a//b` sont-elles équivalentes pour toutes les valeurs entières de `a` et de `b` ?

Le démontrer ou proposer un contre-exemple.

3.1.4 Booléens

Constantes

Les booléens constituent un type spécial dont les constantes sont particulièrement simples : il n'y en a que deux, `True` et `False`. Ils servent à représenter le résultat de l'évaluation d'expressions logiques qui peuvent prendre soit la valeur vraie, représentée par `True`, soit la valeur fausse, représentée par `False`.

Opérateurs sur les booléens

Les opérateurs sur les booléens correspondent aux connecteurs logiques que l'on manipule en mathématiques ou en sciences de l'ingénieur :

- **La négation** ou *non logique*, dont le symbole Python est **`not`**. L'expression **`not`** *`b`* a la valeur `True` si *`b`* s'évalue à `False`, et la valeur `False` si *`b`* s'évalue à `True`.
- **La conjonction** ou *et logique*, dont le symbole Python est **`and`**. L'expression *`b1`* **`and`** *`b2`* a la valeur `True` si *`b1`* et *`b2`* s'évaluent à `True`. Si une des deux expressions s'évalue à `False` alors *`b1`* **`and`** *`b2`* a la valeur `False`.
- **La disjonction** ou *ou logique*, dont le symbole Python est **`or`**. L'expression *`b1`* **`or`** *`b2`* a la valeur `False` si *`b1`* et *`b2`* s'évaluent à `False`. Si une des deux expressions s'évalue à `True` alors *`b1`* **`or`** *`b2`* a la valeur `True`.

Ces règles d'évaluations sont résumées dans le tableau suivant :

b1	*b2*	**not** *b1*	*b1* **and** *b2*	*b1* **or** *b2*
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

En voici quelques exemples :

```
In [33]: True and False
Out[33]: False
```

```
In [34]: not True
Out[34]: False
```

```
In [35]: not (not True)
Out[35]: True
```

Les opérateurs **and** et **or** sont dits *paresseux* : ils ne calculent que ce qui est nécessaire pour évaluer une expression. Par exemple :

```
In [36]: 0 != 0 and 1/0 == 2
Out[36]: False
```

Le booléen situé à gauche de **and** valant `False`, celui de droite n'est pas évalué. Écrite dans l'ordre inverse, l'expression aurait produit une erreur de division par zéro. De même, l'opérateur **or** n'évalue pas son membre droit si son membre gauche vaut `True`.

Précédence des opérateurs sur les booléens

L'opérateur **not** a précédence sur **or** et **and**.

```
In [37]: not True or True
Out[37]: True
```

```
In [38]: not (True or True)
Out[38]: False
```

L'opérateur **and** a précédence sur **or**.

```
In [39]: True or False and False
Out[39]: True
```

```
In [40]: (True or False) and False
Out[40]: False
```

Opérateurs de comparaison

L'apparition la plus fréquente de booléens se fait lors de comparaisons d'autres types.

La comparaison la plus élémentaire est le test d'égalité. L'expression `e1 == e2` s'évalue au booléen `True` si `e1` et `e2` s'évaluent à des valeurs égales, sinon elle s'évalue à `False`.

```
In [41]: 1 == 3 - 2
Out[41]: True
```

```
In [42]: 1 == 0
Out[42]: False
```

Python dispose d'un raccourci pour l'expression `not (e1 == e2)`, à savoir `e1 != e2`.

ATTENTION L'égalité et les flottants

C'est l'occasion de constater que, conformément à ce qui a été vu au chapitre 2, les flottants ne sont pas des représentations exactes des nombres décimaux.

➥

```
In [43]: 0.1 + 0.1 + 0.1 == 0.3
Out[43]: False
```

Tester l'égalité de deux flottants est donc presque toujours une erreur. Il vaut mieux tester si la différence de ces deux flottants est ou non significative, c'est-à-dire est plus petite qu'une précision donnée.

Les types que l'on va considérer dans cet ouvrage sont pour la plupart également comparables pour une relation d'ordre fixée. On peut alors utiliser cet ordre pour comparer deux expressions.

L'ordre strict s'écrit :

```
In [44]: 1 < 3
Out[44]: True
```

ou dans l'autre sens :

```
In [45]: 3 > 1
Out[45]: True
```

Pour effectuer des comparaisons au sens large, on ajoute = après le symbole de comparaison :

```
In [46]: 1 <= 3
Out[46]: True
```

```
In [47]: 1 <= 1
Out[47]: True
```

```
In [48]: 3 >= 1
Out[48]: True
```

Enfin, il est possible d'effectuer des comparaisons avec plus de deux éléments. On écrit alors cela comme en mathématiques :

```
In [49]: 1 < 2 < 3
Out[49]: True
```

On peut également combiner les opérateurs :

```
In [50]: 0 <= 1 < 2 == 4-2 < 5
Out[50]: True
```

ATTENTION Les opérateurs logiques bit à bit

Il existe en Python comme dans d'autres langages deux opérateurs `&` et `|` ressemblant à **and** et **or** mais avec de subtiles différences de signification qui peuvent rendre leur comportement difficile à comprendre :

- Ils ont une plus forte précédence que les comparaisons, ainsi on aura :

```
In [51]: False == False | True
Out[51]: False
```

car dans cette expression on évalue d'abord `False | True`.

- Ils ne sont pas paresseux, les deux membres sont systématiquement évalués :

```
In [52]: (0 != 0) & (1/0 == 2)
ZeroDivisionError: division by zero
```

- Ils désignent en réalité des opérations logiques *bit à bit*. Par exemple, les représentations binaires de `9` et `10` sont respectivement `1001` et `1010`. Si on calcule le « et logique » de chacun de leurs bits de même position, on obtient le mot `1000`, ce qui explique le résultat suivant :

```
In [53]: 9 & 10
Out[53]: 8
```

On évitera donc de se servir de `&` et `|`, à moins de savoir précisément pourquoi on fait ce choix et quelles en seront les conséquences.

Exercice 3.8 Quelle est la valeur des expressions booléennes suivantes, lorsqu'elle existe ?

- `3 * 3.5 > 10`
- `3. * 7 == 21`
- `3 - 1 => 1`
- `0 < 10**-300 == 100**-150`
- `not (2-1 == 1 == 4+3)`
- `not (True and False)`
- `not (not True)`
- `(5.5 * 2 == 11. or 1/2 != .5) and (3 % 2 == 0)`

Vérifier les réponses dans un interpréteur.

Exercice 3.9 Écrire des expressions booléennes traduisant les conditions suivantes. Les nombres mentionnés sont tous des flottants.

- Le point de coordonnées (x, y) est à l'intérieur du cercle de centre (z, t) et de rayon r.
- Les points de coordonnées (x, y) et (z, t) sont situés sur une même droite parallèle à l'un des axes du repère.
- Les points de coordonnées (x, y) et (z, t) sont les sommets opposés d'un carré dont les côtés sont parallèles aux axes du repère.
- Il existe un triangle dont les côtés mesurent respectivement a, b et c.

Exercice 3.10 Écrire des expressions booléennes traduisant les conditions suivantes. Les nombres mentionnés sont tous des entiers.

- L'entier n est divisible par 5.
- Les entiers m et n sont tels que l'un est multiple de l'autre.
- Les entiers m et n sont de même signe.
- Les trois entiers m, n et p sont de même signe.
- n est le plus petit multiple de 7 supérieur à 10^{100}.
- Les trois entiers m, n et p sont distincts deux à deux.

3.2 Variables

3.2.1 Notion de variable

Dans un programme, une *variable* sert à désigner une zone mémoire de l'ordinateur. On peut y stocker une valeur, accéder à cette dernière et la changer.

Pour faire référence à une zone mémoire on utilise un *nom de variable*. On représentera dans ce chapitre et les suivants une variable par un diagramme en forme de rectangle dont le contenu est la valeur de la variable, surmonté d'une étiquette figurant le nom de la variable.

Ainsi la variable nommée x de contenu 42 sera représentée par : x 42 .

Dans certains langages de programmation, une variable peut avoir deux noms, comme y z 3.14 qui a à la fois le nom y et le nom z, mais ce comportement n'est pas possible en Python, excepté dans le cas de structures complexes telles les listes (voir chapitre 6).

On considère qu'une valeur à laquelle n'est attaché aucun nom de variable n'existe pas en mémoire. C'est raisonnable car il s'agit alors d'espace mémoire que le système peut réutiliser ailleurs.

EN PRATIQUE Organisation de la mémoire

Nous avons fait le choix de décrire de façon idéalisée le stockage des valeurs en mémoire et l'attribution des noms de variables car cela suffit à expliquer le comportement de tous les programmes que nous rencontrerons, et ce dans la majorité des langages de programmation.

Une fois que l'on est familier avec la notion de variable, on peut s'intéresser de plus près à la façon précise dont un langage la met en œuvre. Mais il est important, quand on apprend l'informatique, de commencer par acquérir les concepts généraux qui pourront resservir dans un autre contexte, et ensuite seulement de préciser comment ces concepts se traduisent dans un cadre particulier.

3.2.2 État et valeur d'une expression

L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé l'*état*.

De l'état courant dépend donc la valeur de toute expression contenant au moins un nom de variable. Lors de l'évaluation de l'expression, on remplace les noms de variables par leurs valeurs ; on dit qu'on a substitué à la variable sa valeur dans l'expression.

L'expression `x+3` prend ainsi la valeur 5 dans l'état [x: 2] car le nom de variable `x` a été remplacé par son contenu 2. Dans l'état [x: 4], cette expression prend la valeur 7.

Dans l'état [x: 1 | y: 2 | z: 3] qui comporte plusieurs variables, l'expression `x+y*z` s'évalue en `1+2*3` et a donc la valeur `7`.

Au cours de l'évaluation d'une expression, l'état ne peut pas changer. Ainsi, dans l'expression `x*x`, aux deux occurrences du nom `x` sera substituée la même valeur.

Si, lors de l'évaluation d'une expression, un nom de variable est utilisé alors qu'il n'apparaît pas dans l'état courant, il devient impossible d'attribuer une valeur à cette expression. Python renvoie alors le message suivant :

```
In [54]: x
NameError: name 'x' is not defined
```

Pour qu'une expression ait une valeur, il faut donc que chacun des noms de variables qu'elle contient soit présent dans l'état courant. Toutefois, cela n'est pas suffisant. On a déjà vu qu'une expression comme `1 // 0` n'avait jamais de valeur ; donc, l'expression `1 // x` peut ne pas avoir de valeur, par exemple dans l'état [x: 0], alors que `x` est bien présent dans cet état.

Pour aller plus loin **Fonction d'évaluation**

Il est possible de formaliser cette notion d'évaluation d'expression. On note $\mathcal{E}$ l'ensemble des expressions, $\mathcal{D}$ l'ensemble des valeurs possibles (l'union de toutes les constantes de tous les types) et $\mathcal{V}$ l'ensemble des noms de variables. Un état peut alors être décrit sous la forme d'un sous-ensemble de $\mathcal{V} \times \mathcal{D}$, dans lequel deux éléments ne partagent pas le même nom de variable. Cela signifie qu'à chaque nom de variable on associe la valeur correspondante.

L'état [x: 1 | y: 2 | z: 3] correspond alors à l'ensemble $\{(x, 1), (y, 2), (z, 3)\}$.

On note $\mathcal{S}$ l'ensemble des états.

Avec ces notations, l'évaluation est une fonction un peu particulière :

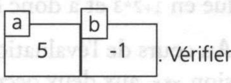
$$\text{eval} : \mathcal{E} \times \mathcal{S} \mapsto \mathcal{D}$$

En effet, comme on vient de le voir, elle n'a pas toujours de valeur à associer à un couple (expression,état). On dit que l'évaluation est une fonction *partielle*.

La définition de eval se fait en raisonnant sur la forme des expressions ; on parle alors de définition par induction. Par exemple, on pourra définir des règles de la forme :

$$\text{eval}(e_1 + e_2, s) = \text{eval}(e_1, s) + \text{eval}(e_2, s)$$

À gauche du symbole =, le symbole + fait référence à l'opérateur de construction des expressions Python. À droite de l'égalité, il s'agit de l'opération mathématique d'addition.

Exercice 3.11 Déterminer de tête la valeur des expressions suivantes dans l'état 

. Vérifier vos réponses à l'aide de l'interpréteur interactif.

- `a + 2`
- `b - 1`
- `a + 2*b`
- `a * a * a`
- `b <= 0`
- `b <= 0 or a < 10`
- `a * b < 2`

Exercice 3.12 * Déterminer dans chacun des cas suivants tous les états tels que :

- `ab + 1` vaut 2.
- `jj / jj` vaut 1.
- `ab * jj` vaut 0.
- Ces trois conditions à la fois sont respectées.

3.2.3 Déclaration et initialisation

Déclarer une variable consiste à l'ajouter à l'état. En Python, cette déclaration est toujours accompagnée d'une *initialisation*. Elles se font en évaluant une instruction (voir chapitre 4) de la forme :

```
nom_de_variable = expression
```

Jusqu'ici, on a uniquement évalué des expressions. Une instruction est une autre forme d'interaction qui demande d'effectuer une modification de l'état. En général, une instruction n'a pas de valeur ; après l'avoir exécutée, Python n'affiche rien. Par exemple :

```
In [55]: x = 2
```

redonne directement l'invite de l'interpréteur, non sans avoir modifié l'état qui contient désormais [x : 2]. Il suffit d'évaluer une expression à la suite pour le constater :

```
In [56]: x + 1

Out[56]: 3
```

Comme on l'a annoncé plus haut, l'instruction :

```
In [57]: y = x
```

fait passer dans l'état [x: 2 | y: 2]. En effet, l'expression *x* a été évaluée à 2 et c'est le résultat de cette évaluation qui est placé dans la variable de nom *y*.

On remarque qu'il n'est pas possible de supprimer une déclaration de l'état. Pour que *x* ne soit plus défini, il n'y a pas d'autre solution que de relancer Python.

> **POUR ALLER PLUS LOIN Suppression de variables**
>
> Il est en fait possible d'accéder à une partie de l'état à l'aide de `locals()` et de supprimer une variable *x* ainsi :
>
> ```
> del locals()['x']
> ```
>
> Il n'est pas recommandé d'utiliser ce genre de mécanisme et il vaut mieux considérer qu'il est impossible de supprimer une variable.

3.2.4 Affectation

Pour changer la valeur d'une variable, on utilise la même instruction que pour la déclaration.

Si on est dans l'état [x: 1], l'instruction :

```
In [58]: x = 2
```

fait passer dans l'état [x: 2].

Une variable peut changer de type par affectation :

```
In [59]: x = 1.2
```

Pour ajouter 1 à la variable *x*, on écrit :

```
In [60]: x = x+1
```

Dans cette instruction, *x* joue deux rôles bien différents. À gauche, il s'agit du nom de la variable sur laquelle s'effectue l'affectation. À droite, il apparaît comme élément de l'ex-

pression qui va être évaluée. Ainsi, si l'état courant est [x : 3], l'évaluation de *x*+1 va donner la valeur 4 et l'affectation est équivalente à *x* = 4. On passe donc dans l'état [x : 4].

EN PRATIQUE Opération-affectation

L'instruction *x* = *x*+1 est équivalente à l'instruction *x* += 1, qu'on peut lire « ajouter 1 à *x* », sous-entendu dans l'état courant. Des opérations-affectations similaires existent pour la plupart des opérateurs courants : par exemple *y* *= *e* multiplie la variable *y* par l'expression *e*, et *z* -= *e* retranche l'expression *e* à la variable *z*.

Il existe une expression particulière, `input()`, qui attend que l'utilisateur tape quelque chose au clavier et qui prend pour valeur la chaîne de caractères correspondante. On l'utilise très souvent sous la forme *variable* = `input()`, de sorte que la variable contiendra ce qui est tapé au clavier.

```
In [61]: a = input()
3.5
```

Précisons que `3.5` est ici ce que l'utilisateur a tapé au clavier ; cette instruction, comme les autres affectations, n'affiche rien. Ensuite on a :

```
In [62]: a
Out[62]: '3.5'
```

On voit que la valeur donnée à *a* est une chaîne de caractères ; si on veut plutôt récupérer une valeur numérique, on écrira `int(input())` ou `float(input())` pour obtenir une valeur respectivement entière ou en virgule flottante. On verra plus tard que cette expression n'est qu'un cas particulier d'un mécanisme plus général.

Une variante consiste à donner une chaîne de caractères en argument à `input` ; cette chaîne sera affichée à l'écran avant de lire ce que tape l'utilisateur. Cela permet par exemple de poser une question pour guider la réponse attendue.

```
In [63]: a = int(input("Combien en voulez-vous ?"))
Combien en voulez-vous ? 42
```

EN PRATIQUE Dans les versions Python 2.x

La fonction `input()` tente de déterminer automatiquement le type de l'expression tapée au clavier. L'inconvénient est qu'il n'est pas possible d'anticiper de quel type sera une expression `input()` au moment de l'écriture du programme !

Savoir-faire Échanger le contenu de deux variables

On se place dans l'état [x : 1 | y : 2] et l'on souhaite placer le contenu de la variable *y* dans la variable *x* et le contenu de la variable *x* dans la variable *y*.

Un premier essai infructueux consisterait à évaluer les instructions :

```
x = y
y = x
```

Cela ne fonctionne pas car après l'évaluation de la première instruction, on se retrouve dans l'état [x : 2 | y : 2] et la seconde instruction nous ramène dans ce même état.

Lorsque l'on écrit ces deux instructions, en se trompant donc, on a une autre vision des choses. On pense en effet que l'expression `x` dans l'instruction `y = x` fait référence à la valeur initiale de `x` avant d'exécuter les instructions. En fait, et c'est assez naturel, on s'imagine que ces deux instructions forment un bloc et qu'elles seront exécutées en même temps. Or, ce n'est pas le cas ; les instructions sont exécutées l'une après l'autre.

La façon la plus naturelle et la plus générale pour placer dans `y` l'ancienne valeur de `x` consiste à la stocker dans une variable auxiliaire :

```
t = x
x = y
y = t
```

On va exécuter ces instructions l'une après l'autre : après la première, on passe dans l'état [x : 1 | y : 2 | t : 1], après la deuxième dans l'état [x : 2 | y : 2 | t : 1] et enfin après la troisième dans l'état [x : 2 | y : 1 | t : 1].

On constate que les valeurs des variables nommées `x` et `y` ont bien été échangées.

Il faut toutefois prendre garde au fait que la variable `t` existe peut-être déjà dans l'état courant. L'essentiel est de choisir une variable qui soit n'est pas présente dans l'état courant, soit ne nous intéresse plus.

On verra page 75 une solution spécifique à Python pour résoudre ce problème de l'échange de deux variables.

Exercice 3.13 On considère un état dans lequel sont définies trois variables de noms *x*, *y* et *z*. Décrire une suite d'instructions permettant de placer :

- le contenu de *x* dans *z*,
- le contenu de *y* dans *x*,
- le contenu de *z* dans *y*.

Exercice 3.14 Partant de l'état initial vide, décrire l'évolution de l'état lors de l'exécution des instructions suivantes, évaluées du haut vers le bas :

- `x = 3`
- `y = 2`
- `x = 1 + y * x`
- `y = y`
- `y = 1.2 - x`

3.3 Types composés

Sont de type composé les valeurs formées de plusieurs valeurs de types plus simples. Par exemple, les couples d'entiers sont de type composé.

De nombreuses constructions sont définies sur tous les types composés.

3.3.1 Les n-uplets

Construction

Un n-uplet, *tuple* en anglais, est une généralisation du concept de couple ou de triplet. On peut voir ce type comme la traduction informatique d'un produit cartésien d'ensembles. Comme en mathématiques, pour construire une expression n-uplet, il suffit de placer des expressions entre parenthèses séparées par des virgules.

Voici un couple composé d'un entier et d'un flottant :

```
In [64]: (1, 2.2)
Out[64]: (1, 2.2)
```

et un triplet d'entiers :

```
In [65]: (0, 1, 2)
Out[65]: (0, 1, 2)
```

Accès aux composantes

Comme pour les autres valeurs, il est possible de stocker un n-uplet dans une variable :

```
In [66]: t = (1, 2.2)
```

On arrive alors dans l'état | t: (1, 2.2) |. On remarque que dans cette instruction, on peut omettre les parenthèses. Ainsi, l'instruction suivante lui est équivalente :

```
In [67]: t = 1, 2.2
```

Pour accéder aux composantes du n-uplet, c'est-à-dire aux sous-valeurs qu'il contient, on utilise l'expression `t[i]` où `i` est le numéro de la composante, on parle de son *indice*.

> ATTENTION **L'indice de la première composante**
> En Python, comme dans la plupart des langages de programmation, on commence à numéroter à partir de 0 et non de 1.

Pour obtenir la première composante du couple, on pourra alors évaluer :

```
In [68]: t[0]
Out[68]: 1
```

et pour la seconde :

```
In [69]: t[1]
Out[69]: 2.2
```

> ATTENTION **Valeurs immuables**
> Les n-uplets sont *immuables*. Cela signifie qu'il n'est pas possible d'affecter de nouvelles valeurs aux composantes. Ainsi l'instruction suivante produit une erreur :
>
> ```
> In [70]: t[0] = 2
> TypeError: 'tuple' object does not support item assignment
> ```

Déconstruction

Il est également possible de déconstruire un n-uplet en affectant simultanément ses composantes à différentes variables.

L'instruction

```
In [71]: x, y = t
```

amène ainsi dans l'état

t	x	y
(1, 2.2)	1	2.2

.

Il est important de remarquer que l'expression n-uplet est évaluée avant de faire les affectations. On peut donc s'en servir pour échanger deux variables d'un coup :

```
In [72]: x,y = y, x
```

En effet, `y, x` s'évalue comme le couple de première composante la valeur de `y` et de seconde composante la valeur de `x`. Une fois ce couple calculé, on effectue les affectations. C'est une façon élégante, mais spécifique à Python, de résoudre le problème posé dans l'encadré de la page 73.

Cas où $n = 0$ ou 1

Il n'existe qu'un 0-uplet, c'est celui qui ne contient rien, il s'écrit `()`. C'est une valeur un peu étrange dont l'utilité est limitée.

Plus courant, un 1-uplet ne contenant que la valeur `v` s'écrit `(v,)`. Ici, la virgule finale est essentielle, car c'est elle qui distingue le 1-uplet `(v,)` de l'expression `(v)` dont la valeur est `v`.

Concaténation

Il est possible de coller un n-uplet et un p-uplet pour obtenir un $(n+p)$-uplet. On parle de concaténation. L'opérateur correspondant en Python est l'opérateur `+`.

```
In [73]: (1,2) + (3,4,5)
Out[73]: (1,2,3,4,5)
```

Ici, si on veut ajouter un seul élément à un n-uplet, il faut faire attention à bien considérer un 1-uplet.

En effet, l'expression suivante produit une erreur :

```
In [74]: (1,2) + 3
TypeError: can only concatenate tuple (not "int") to tuple
```

L'expression correcte est :

```
In [75]: (1,2) + (3,)
Out[75]: (1,2,3)
```

Test d'appartenance

Il est possible de tester si une valeur appartient à un n-uplet à l'aide de l'opérateur `in` :

```
In [76]: 3 in (1,2,3)
Out[76]: True
```

Longueur d'un n-uplet

On obtient la longueur, c'est-à-dire le nombre d'éléments, d'un n-uplet à l'aide de la fonction `len` :

```
In [77]: len( (1,2) )
Out[77]: 2
```

```
In [78]: len( () )
Out[78]: 0
```

3.3.2 Chaînes de caractères : strings

Construction

Le type des chaînes de caractères, *string* en anglais et dans Python, est celui permettant de représenter des textes. On considère dans un premier temps des textes élémentaires, ceux composés d'une unique lettre ; on les appelle les *caractères*.

En Python, les caractères peuvent être n'importe quelle lettre de l'alphabet, mais aussi des symboles, comme les signes de ponctuation :

```
In [79]: 'a'
Out[79]: 'a'
```

```
In [80]: '?'
Out[80]: '?'
```

Une chaîne de caractères est une suite finie de caractères consécutifs, qu'on note entre apostrophes ou guillemets :

```
In [81]: 'Ceci est une chaine'
Out[81]: 'Ceci est une chaine'
```

```
In [82]: "Cela en est une autre"
Out[82]: "Cela en est une autre"
```

La chaîne vide se note `''` ou `""`.

> **POUR ALLER PLUS LOIN Chaîne de caractères ou n-uplet ?**
> Comme les n-uplets, les chaînes de caractères regroupent plusieurs valeurs. D'ailleurs la chaîne `'Bonjour'` et le septuplet `('B','o','n','j','o','u','r')` représentent le même mot. La principale distinction entre ces deux types, c'est qu'une chaîne ne peut contenir *que* des caractères. Ce faisant, on dispose d'opérations spécifiques pour son traitement. Les opérations définies sur les n-uplets sont, en revanche, également disponibles sur les chaînes.

Accès à un caractère

Comme pour les n-uplets, on peut stocker une chaîne dans une variable :

```
In [83]: s = 'Bonjour'
```

et accéder à chacun des caractères à l'aide de la construction `s[i]` :

```
In [84]: s[2]
Out[84]: 'n'
```

Comme les n-uplets, les chaînes sont immuables :

```
In [85]: s[0] = 'A'
TypeError: 'string' object does not support item assignment
```

Concaténation

Comme pour les n-uplets, on concatène deux chaînes à l'aide de l'opérateur + :

```
In [86]: 'Bonjour '+ 'lecteur !'
Out[86]: 'Bonjour lecteur !'
```

Longueur

Comme pour les n-uplets, on utilise `len` pour obtenir la longueur d'une chaîne :

```
In [87]: len('Bonjour')
Out[87]: 7
```

Sous-chaînes

Un ensemble de caractères consécutifs à l'intérieur d'une chaîne s'appelle une sous-chaîne. Ainsi, `'lecteur'` ou `'jour lec'` sont des sous-chaînes de `'Bonjour lecteur !'`. Pour extraire une sous-chaîne de *s*, on écrit `s[i:j]` où *i* est l'indice du premier caractère de la sous-chaîne et *j* est l'indice du dernier caractère **plus un**.

Exemples :

```
In [88]: s = 'Bonjour lecteur !'
```

```
In [89]: s[0:7]
Out[89]: 'Bonjour'
```

```
In [90]: s[8:15]
Out[90]: 'lecteur'
```

Si $j \leqslant i$ il n'y a pas de sous-chaîne correspondante. Python renvoie alors la chaîne vide :

```
In [91]: s[5:2]
Out[91]: ''
```

Si j dépasse la longueur de la chaîne, la sous-chaîne s'arrête au dernier caractère de la chaîne.

Test d'appartenance

Comme pour les n-uplets, l'opérateur `in` sert à tester l'appartenance d'un caractère à une chaîne.

```
In [92]: 'o' in 'Bonjour'
Out[92]: True
```

Il est à noter qu'il est également possible de tester la présence d'une sous-chaîne dans une chaîne avec la même construction :

```
In [93]: 'lecteur' in 'Bonjour lecteur!'
Out[93]: True
```

```
In [94]: 'Bjr' in 'Bonjour lecteur!'
Out[94]: False
```

Conversion vers des types simples

On peut convertir une valeur d'un type simple vers une chaîne de caractères à l'aide de la construction `str(e)`. La chaîne obtenue est la même que celle que Python affiche par évaluation de *e* :

```
In [95]: str(1.2)
Out[95]: '1.2'
```

Il est possible de reconvertir une telle chaîne vers une valeur d'un type simple :

```
In [96]: int('123')
Out[96]: 123
```

```
In [97]: float('1.2')
Out[97]: 1.2
```

```
In [98]: bool('True')
Out[98]: True
```

3.3.3 Listes : une première approche

Les listes seront étudiées dans le chapitre 6. On se contente ici d'une présentation succincte : une liste est un n-uplet dont on peut changer la valeur des composantes.

Pour construire une liste, on remplace les parenthèses par des crochets :

```
In [99]: [1, 2, 3]
Out[99]: [1, 2, 3]
```

Toutes les opérations vues pour les n-uplets sont définies :

```
In [100]: L = [1, 2]
```

```
In [101]: L[1]
Out[101]: 2
```

```
In [102]: L + [3, 4]
Out[102]: [1, 2, 3, 4]
```

Ici cependant, aucune erreur n'apparaît si on change la valeur d'un élément :

```
In [103]: L[0] = 42
```

En évaluant l'expression L on se rend compte que cet élément a bien changé :

```
In [104]: L
Out[104]: [42, 2]
```

SAVOIR-FAIRE Choisir un type composé pour manipuler des données

- Pour représenter du texte, par exemple lu au clavier ou affiché à l'écran, on utilise une chaîne de caractères.
- S'il n'est pas nécessaire de modifier les composantes de la donnée manipulée, on utilise un n-uplet.
- Si on veut pouvoir modifier les composantes individuellement, on utilise une liste. En particulier, si on veut traiter du texte, il vaut mieux le représenter comme une liste de caractères puisque, à la différence d'une chaîne de caractères, une liste n'est pas immuable.

Exercice 3.15 avec corrigé Quel est le type adapté pour représenter les données suivantes :

1 le nom d'une personne ;
2 les coordonnées d'un point mobile dans l'espace ;
3 un numéro de téléphone ;
4 une adresse IP.

1 Une chaîne de caractères : il y a peu de chances de devoir modifier ce nom au cours du programme.
2 Si le point est amené à se déplacer, il faut pouvoir modifier ses coordonnées : on utilise une liste.
3 On pourrait être tenté d'utiliser un entier, mais la gestion du 0 en début de numéro risque de poser problème. Comme un numéro de téléphone n'a pas vocation à être modifié et qu'il est de longueur fixe, on peut utiliser un 10-uplet d'entiers, voire de caractères.
4 Une adresse IP est constituée de quatre entiers entre 0 et 255. On pourra choisir un quadruplet d'entiers ou bien une liste de quatre entiers, selon l'usage que l'on va en faire dans le programme.

3.3.4 Conversions

Il est possible de convertir des n-uplets en listes et réciproquement, comme pour les types simples :

```
In [105]: list( (1, 2, 3) )
Out[105]: [1, 2, 3]
```

```
In [106]: tuple( [1, 2] )
Out[106]: (1, 2)
```

On peut également *éclater* une chaîne et la convertir en la liste de ses caractères :

```
In [107]: list( 'Bonjour')
Out[107]: [ 'B', 'o', 'n', 'j', 'o', 'u', 'r']
```

La conversion d'une chaîne en n-uplet est également possible. En revanche, l'opération inverse n'est pas possible avec une conversion :

```
In [108]: str( [ 'M', 'o', 'i'] )
Out[108]: "[ 'M', 'o', 'i' ]"
```

SAVOIR-FAIRE Convertir une expression d'un type à un autre (suite)

On veut parfois transformer une expression d'un type donné en une expression d'un autre type. Par exemple, si on lit une chaîne de caractères *s* dans un fichier de données et si cette chaîne représente un nombre, il se peut qu'on veuille l'interpréter comme un flottant ou un entier. Il suffit alors d'écrire, respectivement, `float(s)` ou `int(s)` pour demander à Python de convertir cette chaîne en flottant ou en entier.

De manière générale, les fonctions `list`, `tuple`, `str`, `float` et `int` prennent un argument et le transforment dans la mesure du possible en un objet de type liste, n-uplet, chaîne de caractères, flottant et entier respectivement. On a déjà utilisé ce mécanisme pour traiter les valeurs prises par l'expression `input()`.

Exercice 3.16 Écrire une expression qui indique si les 6 voyelles de l'alphabet sont présentes dans une chaîne de caractères *s*.

Exercice 3.17 Écrire une expression qui vérifie si la chaîne de caractères *s* commence par une majuscule et se termine par un point.

Exercice 3.18 * L'expression `tuple(list(t))` a-t-elle toujours la même valeur que *t* ?

Exercice 3.19 * On suppose que l'état comporte une variable *a* dont la valeur est un nombre, mais dont on ne connaît pas le type.

Écrire une expression qui détermine si *a* est un entier ou un flottant en utilisant uniquement les fonctions de conversion et les opérateurs sur les types composés.

4

Instructions : langage minimal de l'algorithmique

Dans ce chapitre, nous ajoutons trois instructions à la déclaration et à l'affectation : la séquence, l'instruction conditionnelle et les boucles. Ces quelques instructions, à elles seules, suffisent à exprimer tous les algorithmes.

4.1 Instructions

4.1.1 Notion d'algorithme

Un *algorithme* est une procédure permettant de résoudre un problème, écrite de façon suffisamment détaillée pour être suivie sans posséder de compétence particulière ni même comprendre le problème que l'on est en train de résoudre.

On compare souvent les algorithmes à des recettes de cuisine. Cette comparaison est correcte au sens où il n'est pas nécessaire de comprendre pourquoi le four doit être à 180°C et pas à 250°C pour réussir la recette ; elle atteint ses limites quand la recette demande des gestes techniques pour lesquels une expérience en cuisine est nécessaire. De la même façon, une notice de montage peut constituer un algorithme permettant de monter un meuble sans savoir comment son assemblage a été conçu, à condition que tous les outils nécessaires soient fournis avec la notice.

Un algorithme a cette autre particularité qu'il permet en réalité de résoudre une *classe* de problèmes similaires, et non pas un problème unique. Ainsi, un algorithme d'addition comme celui que l'on apprend à l'école primaire sert à calculer la somme de n'importe quels nombres décimaux, et on voit mal quelle utilité il aurait s'il ne calculait la somme que de deux nombres fixés une fois pour toutes.

Pour faire fonctionner un algorithme, il faut donc lui fournir des *données* précisant l'instance du problème qu'il devra traiter. En retour, l'algorithme construira un *résultat* répondant à cette instance du problème.

4.1.2 Notion de programme

Les algorithmes ont existé bien avant les ordinateurs, pour réaliser des tâches purement matérielles comme pour résoudre des problèmes très calculatoires. L'exécution d'un algorithme étant complètement déterminée par celui-ci, la difficulté que posait leur utilisation à la main résidait dans le volume de calculs parfois nécessaires et les inévitables erreurs humaines introduites dans le procédé. Avec l'apparition des premiers ordinateurs, il devenait possible de faire exécuter un algorithme de façon beaucoup plus rapide et plus sûre que jamais.

Le prix à payer est de devoir traduire cet algorithme sous une forme non ambiguë et lisible par la machine. Un *programme* est la traduction d'un algorithme dans un langage particulier, à la fois interprétable par la machine et compréhensible par l'homme. Il est constitué d'un assemblage d'instructions, regroupées dans un fichier texte appelé le code source du programme.

L'exécution du programme commence à la première instruction, puis en exécute d'autres en suivant des règles précises. Le parcours des instructions au cours de l'exécution est appelé le *flot d'exécution*.

4.1.3 Langage minimal de l'algorithmique

Dans le chapitre 3, on a vu que l'ensemble des variables définies à un instant donné de l'exécution d'un programme constituait un état de l'exécution du programme. On a également vu qu'il existait une instruction servant à déclarer une variable ou à changer la valeur d'une variable existante.

De manière plus générale, on appelle *instruction* un ordre de modification de l'état courant de l'exécution d'un programme. On sépare les instructions en deux grandes familles :

- d'une part les instructions simples qui manipulent directement l'état courant ;
- d'autre part les instructions composées qui assemblent d'autres instructions et modifient le flot d'exécution en fonction de l'état courant.

Les instructions simples sont la déclaration et l'affectation, regroupées lors de l'étape d'initialisation d'une variable en Python. Les instructions composées sont au nombre de trois :

- **la séquence**, qui exécute deux instructions l'une à la suite de l'autre ;
- **le test**, ou instruction conditionnelle, qui sert à n'exécuter une instruction que dans certains états ;
- **la boucle**, qui exécute plusieurs fois la même instruction dans un programme.

Il est remarquable que ces cinq instructions, à elles seules, suffisent à exprimer *tous* les algorithmes imaginables. Bien sûr, une telle affirmation demande à être précisée, surtout dans la mesure où on n'a pas donné de définition formelle de ce qu'était un algorithme. C'est là l'objet de la thèse de Church-Turing, qui affirme que tout procédé de calcul pouvant être décrit de façon systématique peut l'être avec ces cinq instructions.

4.1.4 Entrées/sorties

Dans un langage de programmation, on appelle *entrées/sorties* les constructions qui interrompent le flot d'exécution du programme pour communiquer avec l'utilisateur, donnant ainsi un aspect interactif au programme.

On a vu au chapitre 3 qu'il existe une expression particulière, `input()`, qui a pour effet d'interrompre le déroulement du programme, d'attendre que l'utilisateur tape quelque chose au clavier, et de prendre cette valeur. La réciproque existe, sous la forme d'une instruction un peu à part, l'instruction `print`, qui affiche à l'écran les expressions qui lui sont données en argument. Elle ne manipule donc pas l'état, au sens où elle ne modifie pas les contenus des variables, mais seulement l'aspect de l'écran. On s'en sert souvent pour afficher des chaînes de caractères, mais elle est capable d'afficher des expressions quelconques, voire plusieurs expressions à la suite :

```
In [1]: print("Bonjour")
Bonjour
```

```
In [2]: b=42
```

```
In [3]: print(b+3)
45
```

```
In [4]: print("La variable b contient", b, "et rien de plus.")
La variable b contient 42 et rien de plus.
```

Par défaut, `print` écrit les différentes expressions qui lui sont données en arguments sur la même ligne, séparées par des espaces, puis revient à la ligne. On peut les séparer par une autre chaîne de caractères en ajoutant un argument de la forme `sep='...'`, et modifier la fin de ligne par un argument de la forme `end='...'`. Ainsi, l'instruction suivante ne revient *pas* à la ligne (la chaîne *end* est vide).

```
In [5]: print(1, 2, 3, sep='*', end='')
1*2*3
```

EN PRATIQUE **Dans les versions Python 2.x**

L'instruction `print` s'utilise de la même façon, sauf qu'on ne met pas de parenthèses autour de ce qu'on veut afficher. Mettre des parenthèses inutiles dans les instructions `print` d'un programme Python 2.x ne produira pas d'erreur, mais l'affichage obtenu ne sera pas tout à fait celui voulu.

ATTENTION **Afficher la valeur des expressions au cours de l'exécution d'un programme.**

Dans un interpréteur interactif, il suffit de taper une expression pour afficher sa valeur. L'instruction `print` y présente donc peu d'intérêt.

Dans un programme, en revanche, une ligne constituée uniquement d'une expression n'a aucun effet visible et il faut obligatoirement utiliser `print` si on veut afficher un résultat lors de l'exécution.

4.1.5 Séquence d'instructions

La manière la plus simple d'assembler deux instructions consiste à les placer l'une à la suite de l'autre. L'exécution du programme commencera par la première instruction et passera ensuite à la seconde.

On considère le programme suivant :

```
x = x + 1
x = x * 2
```

Si on part de l'état [x : 1], après exécution de la première instruction on passe dans l'état [x : 2], puis après la seconde dans l'état [x : 4].

L'importance de l'ordre dans lequel sont exécutées les instructions se confirme : si on inverse l'ordre des deux instructions, ce programme produit l'état | x : 3 |.

```
x = x * 2
x = x + 1
```

POUR ALLER PLUS LOIN Sémantique de la séquence

Si on note $i(e)$ l'état obtenu en exécutant l'instruction i dans l'état e, on a, après l'exécution des deux instructions i_1 puis i_2, l'état $i_2(i_1(e))$.

Enchaîner deux instructions en séquence revient donc à composer (au sens des fonctions mathématiques) leurs deux actions sur l'état.

Il est possible de considérer des séquences de plus de deux instructions ; on parle alors plus généralement de *bloc d'instructions*.

Exercice 4.1 Les suites de symboles suivantes sont-elles des instructions ou des expressions ?

- `a`
- `b = a`
- `float(input())`
- `b = b - 1`
- `c * 2`
- `print(a + b)`
- `x = int(input())`
- `a == 3`
- `a == b or c == 1`

Exercice 4.2 Décrire l'évolution de l'état au cours de l'exécution du programme ci-dessous, en partant de l'état | b : 5 |.

```
a = 2
a = b - 7
b = b + 1
c = a - b
a = (a-1) / (a+1)
```

Exercice 4.3 * On s'intéresse à l'effet du programme ci-dessous.

```
a = 2*a - b
b = a + b
a = b - a
b = b // 2
```

1 Décrire l'évolution de l'état au cours de l'exécution de ce programme pour différentes valeurs initiales de *a* et *b*.
2 Formuler une conjecture sur les valeurs finales de *a* et de *b* en fonction de leurs valeurs initiales.
3 Démontrer cette conjecture dans le cas où les valeurs initiales de *a* et *b* sont des entiers relatifs.
4 Que penser de cette conjecture dans le cas où *a* et *b* contiennent initialement des nombres en virgule flottante ? Et si on remplace la dernière ligne par `b = b / 2` ?

Exercice 4.4 * Démontrer que la séquence est une opération associative, autrement dit qu'à partir d'un même état e, les instructions « (i_1 suivie de i_2) suivie de i_3 » et « i_1 suivie de (i_2 suivie de i_3) » produisent le même état e'.
En quoi cette propriété légitime-t-elle la notion de bloc d'instructions ?

4.2 Instructions conditionnelles

4.2.1 Test simple

Une instruction conditionnelle n'est exécutée que si une condition donnée est vérifiée par l'état courant.

Pour traduire cela, on utilise l'instruction `if`, qui a en Python la syntaxe suivante :

```
if condition:
    bloc_d_instructions
```

Le bloc d'instructions est exécuté uniquement si `condition` est vérifiée. Dans l'exemple qui suit, on ajoute 1 à la variable `x` seulement si, dans l'état courant, elle a une valeur impaire ; si `x` a une valeur paire, l'instruction d'ajout est ignorée :

```
if x % 2 == 1:
    x = x + 1
```

4.2.2 Indentation signifiante

On a déjà expliqué qu'un bloc est une suite d'instructions qui s'exécutent les unes après les autres. Pour identifier sans ambiguïté les instructions appartenant au bloc du `if`, il est nécessaire de les indenter.

Indenter signifie ajouter un même nombre d'espaces devant chacune des lignes définissant les instructions. Alors que dans certains langages cette pratique n'est qu'une recommandation, elle est une obligation dans Python.

Pour s'y conformer aisément, on peut se fixer pour convention qu'une indentation est constituée de quatre espaces. Le niveau d'indentation d'une ligne correspond alors au nombre d'espaces en début de ligne divisé par quatre.

```
# cette ligne a un niveau d'indentation de 0
    # cette ligne a un niveau d'indentation de 1
        # cette ligne a un niveau d'indentation de 2
```

Au sein d'un bloc, le niveau d'indentation doit être le même. Le code suivant est valide et la ligne `y = y // x` ne s'exécute que si `x` est non nul :

```
if x != 0:
    y = 3
    y = y // x
```

mais celui-ci ne l'est pas :

```
if x != 0:
    y = 3
        y = y // x
```

Lorsque l'on tente de le faire exécuter par Python, on obtient l'erreur suivante :

```
File ``fichier.py'', line 3
  y = y // x
  ^
  IndentationError: unexpected indent
```

La première instruction qui suit une instruction conditionnelle et qui est placée au même niveau d'indentation que l'instruction `if` marque la fin du bloc. En effet, seules les instructions indentées font partie du bloc.

Ici, la dernière instruction s'exécute toujours après le `if` :

```
if x != 0:
    y = 3
y = y // x
```

En cas de `if` au sein d'un `if`, le second bloc doit avoir un niveau d'indentation encore supérieur. Par exemple :

```
if x != 0:
    y = x * x
    if y % 2 == 0:
        y = y + 1
    x = x + y
```

Là, l'instruction `y = y + 1` n'est exécutée que si la seconde condition est vérifiée *en plus* de la première.

Exercice 4.5 Quel est le résultat de l'exécution des instructions suivantes dans l'état

a	b	c
0	2	1

?

```
if a == 0:
    b = 4
else:
    c = 5
b = 1
```

et

```
if a == 0:
    b = 4
else:
    c = 5
    b = 1
```

4.2.3 Test avec alternative

On a vu au chapitre 3 qu'il était possible d'obtenir la négation d'un élément du type `bool` à l'aide de l'opérateur `not`. On pourrait penser que cela permet d'exécuter certaines instructions lorsqu'une condition est vérifiée et d'autres quand elle ne l'est pas, avec deux tests.

Par exemple, si l'on voulait ajouter 1 à *x* quand sa valeur est impaire et la diviser par 2 sinon, on pourrait écrire :

```
if x % 2 == 1:
    x = x + 1
if not(x % 2 == 1):
    x = x // 2
```

Cependant, cet extrait de programme pose de multiples problèmes. On peut déjà remarquer qu'il est redondant, puisque le programmeur doit écrire deux fois la même condition. Il est également inefficace : cette condition devra être évaluée deux fois, ce qui ne pose pas de problème dans l'exemple donné, mais peut prendre un temps de calcul non négligeable dans d'autres cas.

Enfin et surtout, l'action de ce programme sur l'état n'est pas celui que l'on voulait. En effet, dans le cas où la valeur de *x* est impaire, la première instruction conditionnelle ajoute 1 à *x*. Dans le nouvel état obtenu, *x* devient pair. La seconde instruction conditionnelle s'exécute dans ce nouvel état et la condition `not(x % 2 == 1)` s'évalue à `True`. Par conséquent, pour une valeur initiale impaire de *x*, le contenu de *x* est remplacé non pas par `x+1` mais par `(x+1)//2`.

Pour ces trois raisons, on enrichit la syntaxe des tests pour proposer une alternative :

```
if condition:
    bloc_d_instructions_si_la_condition_est_vérifiée
else:
    bloc_d_instructions_si_la_condition_n_est_pas_vérifiée
```

Le programme correct pour l'exemple précédent s'écrit alors :

```
if x % 2 == 1:
    x = x + 1
else:
    x = x // 2
```

SAVOIR-FAIRE Documenter un programme

Dès que l'on écrit un programme de plus d'une dizaine de lignes, il devient indispensable de le rendre parfaitement lisible, pour permettre à un autre programmeur de le comprendre, ou pour le reprendre soi-même plus tard.

Deux bonnes habitudes sont à prendre très tôt, qui sont déterminantes pour la lisibilité d'un programme :

- choisir des noms de variables parlants ;
- ajouter des *commentaires* dans les programmes, autrement dit des lignes écrites en langue naturelle que la machine ne cherche pas à interpréter comme des instructions et qui expliquent le rôle des différentes parties du programme.

Pour préciser qu'une ligne est un commentaire, on la fait précéder d'un symbole particulier. En Python, il s'agit d'un dièse `#`. Si l'on veut écrire un commentaire sur plusieurs lignes, il faut faire précéder chacune d'entre elles de ce symbole ou entourer le commentaire de trois guillemets doubles `"""`. Ces commentaires doivent donner des informations supplémentaires sur le sens du programme. Inutile de commenter en disant par exemple « ceci est une boucle » ; mieux vaut expliquer le rôle de cette boucle.

Exercice 4.6 avec corrigé Le programme ci-dessous affiche l'équation réduite de la droite d'équation cartésienne $ax + by + c = 0$. En proposer une version plus lisible et documentée.

```
if b == 0:
    if a != 0:
        print("x =", -c/a)
else:
    m = -a/b
    p = -c/b
    print("y =", m, "x +", p)
```

Les variables `m` *et* `p` *peuvent être renommées respectivement en* `pente` *et* `coefficient_directeur`*.*
En revanche, les notations a, b et c sont très courantes pour les coefficients d'une équation cartésienne, qui n'ont pas de nom particulier. On peut tout de même donner une indication sur ce que représentent les variables correspondantes dans ce programme, sous forme d'un commentaire en début de programme.
Enfin, il peut être utile d'indiquer rapidement à quoi correspondent chacun des cas testés. On obtient le programme suivant.

```
# a x + b y + c = 0 est l'équation cartésienne d'une droite
if b == 0:
    # droite parallèle à l'axe des ordonnées
    if a != 0:
        # sinon ce n'est pas une droite
        print("x =", -c/a)
else:
    # cas général
    pente = -a/b
    coefficient_directeur = -c/b
    print("y =", pente, "x +", coefficient_directeur)
```

Exercice 4.7 Que penser de ce qu'affiche le programme précédent dans les cas où la pente vaut 0 ou 1 et dans le cas où le coefficient directeur vaut 0 ?
Modifier ce programme pour qu'il produise un affichage plus agréable.

4.2.4 Tests imbriqués

Comme on a pu le voir dans la section sur l'indentation, il est possible de réaliser une instruction conditionnelle dans une instruction conditionnelle. On parle alors de *tests imbriqués*.

Lorsque les tests imbriqués ne servent qu'à séparer une situation en plus de deux cas de figure, l'indentation obligatoire en Python peut rendre difficile la compréhension du programme.

Par exemple, le programme suivant exécute trois instructions différentes suivant la valeur de `x % 3` :

```
if x % 3 == 0:
    x = x + 1
else:
    if x % 3 == 1:
        x = x - 1
    else:
        x = 2 * x
```

Il n'y pas de raison pour que l'instruction `x = x - 1` soit plus imbriquée que l'instruction `x = x + 1`. En effet, elle correspond juste au cas où `x % 3` vaut 1. Il est possible de mettre toutes ces instructions au même niveau en contractant le **else** et le **if** qui le suit : on utilise le mot-clé **elif**. Ainsi, le programme précédent devient :

```
if x % 3 == 0:
    x = x + 1
elif x % 3 == 1:
    x = x - 1
else:
    x = 2 * x
```

Cette fois, les trois instructions sont au même niveau.

SAVOIR-FAIRE Écrire des instructions conditionnelles avec alternatives, éventuellement imbriquées

1 On identifie les cas qui vont demander un traitement différent.
2 S'il n'y a que deux cas, on écrit une condition booléenne permettant de les distinguer et on construit l'instruction `if...else` avec cette condition.
3 S'il y a plus de deux cas, écrire des conditions booléennes traduisant chacun de ces cas. Idéalement, ces conditions sont mutuellement exclusives et la dernière peut donc être ignorée puisqu'elle est équivalente à la négation de toutes les autres. Construire l'instruction `if...elif...else` avec ces conditions.
4 Remplir les blocs d'instructions correspondant à chacun des cas en n'oubliant pas de respecter l'indentation. Si le cas du **else** ne contient aucune instruction, on peut l'omettre complètement.

5 Chacun des cas construits peut être lui-même écrit à l'aide d'une instruction conditionnelle, ce qui permet de structurer l'écriture du programme.
6 Ne pas oublier de revenir au même niveau d'indentation que l'instruction `if` pour la première instruction qui suit l'instruction conditionnelle.

Exercice 4.8 avec corrigé Écrire un programme qui détermine si un automobiliste est en excès de vitesse, connaissant sa vitesse et le type de voie sur lequel il circule (donné sous forme d'une chaîne de caractères).

Les cas à distinguer pour le type de voie sont par exemple `'agglomération'`, `'route'` *et* `'autoroute'`*; il serait bien entendu possible d'en prévoir d'autres. Grâce au mot-clé* `else`*, il n'est pas nécessaire de citer explicitement le dernier cas, mais on peut mettre un commentaire pour expliciter à quoi il correspond.*

Ensuite, dans chacun des cas, il faut distinguer si l'automobiliste dépasse la vitesse limite autorisée ou non. On peut donc écrire le programme suivant.

```
if voie == 'agglomération':
    if vitesse > 50:
        exces = True
    else:
        exces = False
elif voie == 'route':
    if vitesse > 90:
        exces = True
    else:
        exces = False
else:  # voie == 'autoroute'
    if vitesse > 130:
        exces = True
    else:
        exces = False
```

On remarque cependant que l'instruction conditionnelle interne est toujours de la même forme et on aura donc ici plutôt intérêt à utiliser une variable pour ne pas avoir à la répéter.

```
if voie == 'agglomération':
    limite = 50
elif voie == 'route':
    limite = 90
else:  # voie == 'autoroute'
    limite = 130

if vitesse > limite:
    exces = True
else:
    exces = False
```

La seconde instruction conditionnelle n'est ici plus imbriquée dans la première, elle doit donc être placée au même niveau d'indentation.

Enfin, on peut remarquer que, puisqu'on ne cherche qu'à mettre une valeur booléenne dans la variable `exces`*, la dernière instruction conditionnelle pourrait être remplacée par l'affectation* `exces = vitesse > limite`*.*

Savoir-faire Comprendre un algorithme et expliquer ce qu'il fait

On identifie le rôle de chacune des variables utilisées. Si nécessaire, on peut dérouler à la main une exécution du programme en notant l'état au fur et à mesure pour visualiser l'évolution des variables.

Exercice 4.9 avec corrigé Que fait ce programme ?

```
if a>b:
    if a>c:
        m = a
    else:
        m = c
else:
    if b>c:
        m = b
    else:
        m = c
```

Les contenus des variables `a`, `b` et `c` ne sont pas modifiés dans ce programme : il s'agit de données à partir desquelles on calcule une valeur, ici dans la variable `m`.

Dérouler ensuite le programme pour quelques valeurs de `a`, `b` et `c` permet rapidement de conjecturer qu'à la fin de l'exécution, `m` contient le maximum des trois valeurs. Il serait judicieux d'appeler plutôt cette variable `max` pour rendre son rôle évident et éviter par exemple la confusion avec un minimum.

Savoir-faire Modifier un programme existant pour obtenir un résultat différent

L'intérêt de partir d'un programme existant est qu'il n'est pas toujours nécessaire d'en comprendre le fonctionnement en détail pour l'adapter à un nouveau besoin. Il importe avant tout :

- d'identifier les parties du programme qui doivent être modifiées et celles qui sont à conserver ;
- de les modifier en conséquence ;
- éventuellement d'adapter les entrées et les sorties au nouveau programme ;
- et, comme toujours, de le tester sur des exemples bien choisis.

Exercice 4.10 avec corrigé Le programme suivant détermine si un individu est en surpoids en calculant son indice de masse corporelle.

```
masse = float(input("Quelle est votre masse en kg ?"))
taille = float(input("Quelle est votre taille en m ?"))
imc = masse / taille**2
if imc > 25:
    print("Vous etes en surpoids.")
```

Adapter ce programme pour qu'il détermine également si l'individu est en sous-poids, ce qui correspond à un indice de masse corporelle inférieur à 18.

Il n'est pas nécessaire ici de connaître la définition de l'indice de masse corporelle pour comprendre le déroulement du programme ni pour faire la modification voulue. En effet, on va réutiliser le calcul effectué à la troisième ligne sans avoir besoin de le modifier. Ensuite, on peut s'inspirer des deux dernières lignes pour écrire le cas du sous-poids, qui est très similaire à celui du surpoids.

Il suffit donc d'ajouter les deux lignes suivantes à la fin du programme :

```
if imc < 18:
    print("Vous etes en sous-poids.")
```

SAVOIR-FAIRE Mettre un programme au point en le testant

Pour vérifier si un programme ne produit pas d'erreur au cours de son exécution et s'il effectue réellement la tâche que l'on attend de lui, une première méthode consiste à exécuter plusieurs fois ce programme, en lui fournissant des entrées, appelées tests, qui servent à détecter les erreurs éventuelles. Pour qu'elles jouent leur rôle, il faut choisir ces entrées de sorte que :

- on sache quelle doit être la sortie correcte du programme avec chacune de ces entrées ;
- chaque cas distinct d'exécution du programme soit parcouru avec au moins un choix d'entrées ;
- les cas limites soient essayés : nombres nuls ou négatifs, liste vide, etc.

Exercice 4.11 avec corrigé Proposer des jeux de tests pour le programme de l'exercice 4.9.

Comme on calcule ici le maximum des trois nombres `a`, `b` *et* `c`, *il faut au moins tester tous les ordres possibles pour ceux-ci. Cela donne déjà 6 cas distincts pour le triplet* `(a,b,c)` *:* $\{(1,2,3);(1,3,2);(2,1,3);(2,3,1);(3,1,2);(3,2,1)\}$.

De plus, il pourrait être intéressant de vérifier ce qui se passe dans des cas particuliers comme celui où `a`, `b` *et* `c` *sont égaux.*

Exercice 4.12 Modifier le programme de l'exercice 4.9 pour qu'il détermine le minimum des trois valeurs `a`, `b` et `c`.

Exercice 4.13

1 Écrire un programme qui calcule la négation d'une variable booléenne sans utiliser l'opérateur **`not`**.
2 Écrire un programme qui calcule le *et logique* de deux variables booléennes sans utiliser l'opérateur **`and`**.
3 Écrire un programme qui calcule le *ou logique* de deux variables booléennes sans utiliser l'opérateur **`or`**.

Exercice 4.14 La fonction **`random.random()`** renvoie un nombre à virgule flottante pseudo-aléatoire compris entre 0 inclus et 1 exclu. Pour utiliser cette fonction, il suffit d'exécuter l'instruction **`import random`** dans l'interpréteur interactif, ou de la placer en début de programme.

À l'aide de **`random.random()`**, écrire un programme qui simule la loi uniforme sur l'intervalle $[a;b]$, où a et b sont deux réels donnés.

Exercice 4.15 Toujours à l'aide de la fonction **`random.random()`**, écrire un programme qui choisit la valeur booléenne `True` ou `False` de façon équiprobable.

Exercice 4.16 Toujours à l'aide de la fonction **`random.random()`**, écrire un programme qui simule la loi uniforme sur l'intervalle d'entiers $[\![a;b]\!]$, où a et b sont deux entiers donnés.

4.3 Boucles conditionnelles

4.3.1 Nécessité des boucles

Maintenant que l'on dispose d'instructions conditionnelles, on va essayer de réaliser un programme qui, à partir d'un état contenant une variable entière `c` de valeur n, définit une nouvelle variable `p` de valeur 2^n et place la valeur 0 dans `c`.

Si on suppose que n est compris entre 0 et 2, on pourrait écrire le programme suivant :

```
p = 1
if c != 0:
    p = p * 2
    c = c - 1
    if c != 0:
        p = p * 2
        c = c - 1
```

En effet, si `c` contient la valeur 0, le test principal s'évalue à `False` et `p` contient la valeur 1. Si `c` contient la valeur 1, seul le premier test est vérifié car à l'issue de l'instruction `c = c - 1`, la variable `c` contient la valeur 0. Pour que ce programme fonctionne lorsque n est compris entre 0 et 3, il suffit d'ajouter à nouveau un test :

```
p = 1
if c != 0:
    p = p * 2
    c = c - 1
    if c != 0:
        p = p * 2
        c = c - 1
        if c != 0:
            p = p * 2
            c = c - 1
```

On voit bien les limites d'une telle définition. Pour que cela soit valable avec des valeurs quelconques de `c`, il faudrait écrire un programme contenant une infinité de lignes (littéralement ici, car les entiers en Python sont non bornés).

On remarque pourtant que les instructions à effectuer après le test `c != 0` sont toujours les mêmes. On aimerait donc exécuter ces instructions *tant que* le test est vérifié. C'est exactement ce que l'on appelle une *boucle conditionnelle*.

Le terme boucle fait référence au flot d'exécution. Celui-ci boucle autour des instructions à exécuter tant que la condition est vérifiée.

4.3.2 Syntaxe d'une boucle conditionnelle

En Python, on écrit une boucle conditionnelle ainsi :

```
while condition:
    bloc_d_instructions
```

Les contraintes d'indentation pour le bloc d'instructions sont les mêmes que pour une instruction conditionnelle et la fin du bloc est marquée par le retour au niveau d'indentation du `while`. Ce bloc d'instructions est appelé le corps de boucle et chaque passage dans ce bloc est appelé une itération. Lorsqu'on effectue la première itération, on dit qu'on entre dans la boucle ; on en sort lorsque l'on a fini d'exécuter la dernière itération.

Le programme calculant 2^n deviendra alors :

```
p = 1
while c > 0:
    p = p * 2
    c = c - 1
```

On va détailler l'exécution de ce programme à partir de l'état [c : 3] :

- Avant la première itération : [c : 3] [p : 1]
- Après la première itération : [c : 2] [p : 2]
- Après la deuxième itération : [c : 1] [p : 4]
- Après le troisième itération : [c : 0] [p : 8]

Ce dernier état est l'état final, car la condition `c > 0` n'est plus vérifiée et on sort donc de la boucle.

Exercice 4.17 Exécuter ce programme à l'aide d'un débogueur et vérifier que l'évolution des variables est bien celle prévue.

Exercice 4.18 Modifier ce programme pour qu'il calcule k^n où k et n sont deux entiers naturels quelconques pouvant être choisis par l'utilisateur.

Savoir-faire Écrire un programme utilisant une boucle `while`

1 On identifie la condition de la boucle ; il est souvent plus commode de chercher une condition de *sortie*, puis de calculer sa négation ou tout simplement d'utiliser l'opérateur `not`.

2 On écrit le corps de la boucle, en s'assurant que celui-ci modifiera la valeur de la condition à certaines itérations.

3 On prévoit une initialisation des variables en amont de la boucle.

4 Il est parfois nécessaire de faire un dernier traitement à la suite de la boucle ; dans tous les cas, on n'oubliera pas de revenir au niveau d'indentation du `while`.

Exercice 4.19 avec corrigé Déterminer le rang du dernier terme strictement positif de la suite récurrente définie par $u_{n+1} = \frac{1}{2}u_n - 3n$, la valeur de u_0 étant donnée dans la variable *u_0*.

On utilisera dans ce programme les variables n *pour le rang courant et* u *pour la valeur de* u_n.

1 *Le calcul des termes* u_n *devra s'arrêter dès que* $u_n \leqslant 0$ *; en prenant la négation de cette expression et en la traduisant avec les variables du programme, on trouve la condition* u > 0.

2 *Le corps de la boucle consiste simplement à calculer la valeur de* u_{n+1} *à partir de celle de* u_n *et à mettre à jour le rang. Les deux lignes ci-après ne doivent pas être interchangées sinon la formule de récurrence n'est pas correctement traduite.*

```
u = 0.5*u - 3*n
n = n + 1
```

La variable u *étant modifiée, la valeur de la condition* u > 0 *pourra changer au cours de l'exécution de la boucle.*

3 *En amont de la boucle, il faut initialiser les variables* u *et* n*.*

```
u = u_0
n = 0
```

4 *Enfin, à la sortie de la boucle, on a atteint le premier terme* u_n *négatif ou nul, mais il était demandé le rang du dernier terme strictement positif. Il faut donc « revenir en arrière » d'un rang.*

```
n = n - 1
```

Voici le programme complet :

```
u = u_0
n = 0
while u > 0:
    u = 0.5*u - 3*n
    n = n + 1
n = n - 1
```

4.3.3 Terminaison de boucle

Compteur

Dans le programme précédent, on remarque que la variable *c* joue le rôle d'un compteur.

Au départ, *c* contient le nombre d'itérations à effectuer. Après chaque itération, on enlève un à ce nombre. La condition d'arrêt de la boucle teste si les n itérations ont été faites.

Mathématiquement, il est garanti que l'on sorte de la boucle car :

- la valeur de *c* est un entier strictement positif ;
- elle décroît strictement après chaque itération.

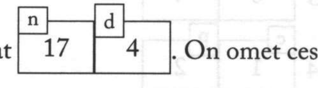
Comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations.

Exemple de la division euclidienne

Il n'est pas nécessaire d'avoir une variable du type compteur de boucle. Il suffit qu'une quantité vérifie bien ces deux propriétés : être un entier positif tout au long de l'algorithme et décroître strictement après chaque itération. On appelle cette quantité un *variant de boucle*.

Pour illustrer cela, on considère maintenant un programme réalisant l'algorithme usuel de division euclidienne pour des entiers naturels. Ici, l'état initial contient deux variables n et d et, à l'issue de l'exécution, on souhaite que les deux variables q et r contiennent le quotient et le reste dans la division euclidienne de la valeur de n par la valeur de d.

```
q = 0
r = n
while r >= d:
    q = q + 1
    r = r - d
```

On va détailler l'exécution de ce programme à partir de l'état 

. On omet ces deux variables, qui ne sont pas modifiées.

- Avant la première itération :

q	r
0	17

- Après la première itération :

q	r
1	13

- Après la deuxième itération :

q	r
2	9

- Après la troisième itération :

q	r
3	5

- Après la quatrième itération :

q	r
4	1

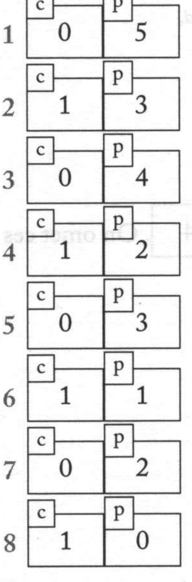
Ici, si la valeur de d est un entier strictement positif, la variable r reste positive tout au long de l'algorithme et, après chaque itération, elle diminue de la valeur de d, donc elle décroît strictement. Ainsi ce programme se termine-t-il.

Cas où l'expression qui décroît n'est pas une variable

On considère le programme suivant :

```
c = 0
while p > 0:
    if c == 0:
        p = p - 2
        c = 1
    else:
        p = p + 1
        c = 0
```

Partant de l'état initial [p: 5], on obtient successivement les états suivants :



On note p_i et c_i les valeurs des variables `p` et `c` après l'itération i.

On considère alors l'expression $2p_i+3c_i$. Comme p_i et c_i sont toujours des entiers naturels, cette expression est bien un entier naturel. On étudie l'effet de la $i+1$-ème itération sur l'état.

- Si $c_i = 0$ alors $p_{i+1} = p_i - 2$ et $c_{i+1} = 1$, d'où

$$2p_{i+1} + 3c_{i+1} = 2p_i - 4 + 3 = 2p_i - 1 < 2p_i = 2p_i + 3c_i$$

- Sinon, on a $c_i = 1$, $p_{i+1} = p_i + 1$ et $c_{i+1} = 0$. Ici encore, on a

$$2p_{i+1} + 3c_{i+1} = 2p_i + 2 < 2p_i + 3 = 2p_i + 3c_i$$

Dans tous les cas, la quantité $2p_i + 3c_i$ diminue strictement à chaque itération et ainsi la boucle se termine.

SAVOIR-FAIRE Démontrer qu'une boucle se termine effectivement

On identifie un variant, autrement dit une expression (c'est souvent le simple contenu d'une variable) :

- qui est un entier positif tout au long de la boucle,
- et qui diminue strictement après chaque itération.

On peut alors en conclure que la boucle se termine.

Exercice 4.20 Démontrer la terminaison du programme écrit à l'exercice 4.19.

4.3.4 Invariant de boucle

Lorsque l'on a écrit un programme, il reste à vérifier qu'il est correct, c'est-à-dire qu'il calcule bien ce qu'on attend ; on peut tester quelques cas significatifs, mais il est beaucoup plus satisfaisant de *démontrer* qu'il est correct dans tous les cas. Une des manières les plus efficaces de le faire est d'établir un *invariant de boucle*, c'est-à-dire une propriété qui est vérifiée tout au long de l'exécution d'une boucle. Cette démarche est à rapprocher du raisonnement par récurrence : en ne s'intéressant qu'aux valeurs initiales des variables et à leur évolution au cours d'une seule itération, on peut en déduire des propriétés valides quel que soit le nombre d'itérations.

SAVOIR-FAIRE Démontrer qu'une boucle produit l'effet attendu au moyen d'un invariant

On utilise une invariant de boucle, c'est-à-dire une propriété qui :

- est vérifiée avant d'entrer dans la boucle,
- si elle est vérifiée avant une itération, est vérifiée après celle-ci,
- lorsqu'elle est vérifiée en sortie de boucle permet d'en déduire que le programme est correct.

Exercice 4.21 avec corrigé Démontrer que le programme de calcul de 2^n est correct, c'est-à-dire que lorsque l'exécution se termine, la variable `p` contient bien la valeur 2^n où $n \geqslant 0$ est la valeur initiale de la variable `c`.

Pour raisonner, on note c_i et p_i les valeurs des variables `c` et `p` après l'exécution de la i-ème itération. L'état au moment de l'entrée dans la boucle est

$$c_0 = n \qquad p_0 = 1$$

De plus, le corps de la boucle assure les relations suivantes pour toute itération i :

$$c_{i+1} = c_i - 1 \qquad p_{i+1} = 2p_i$$

On remarque alors que la propriété suivante est toujours vérifiée : pour toute itération i, on a $p_i = 2^{n-c_i}$ et $c_i \geqslant 0$.
En effet :
- *Elle est vérifiée pour $i = 0$: $p_0 = 1 = 2^0 = 2^{n-c_0}$ et $c_0 = n \geqslant 0$.*
- *Si on a $p_i = 2^{n-c_i}$ et si l'on effectue une itération de plus, on a alors :*

$$p_{i+1} = 2p_i = 2^{n-c_i+1} = 2^{n-c_{i+1}} \quad \text{et} \quad c_{i+1} = c_i - 1$$

Puisqu'on a effectué l'itération, $c_i > 0$ et donc $c_{i+1} \geqslant 0$.
Toutefois, lorsque la condition `c > 0` n'est plus vérifiée, en sortie de boucle, on a $c_i = 0$ et donc $p_i = 2^n$.

Toute la subtilité de ce raisonnement tient bien entendu dans le choix du bon invariant.

Exercice 4.22 * Démontrer la correction du programme de division euclidienne présenté page 99 à l'aide d'un invariant de boucle.

4.3.5 Boucle infinie

On a vu comment démontrer qu'une boucle se termine en un nombre fini d'étapes. On étudie ici ce qui se passe quand ce n'est pas le cas.

On considère la boucle suivante, qui est une variante du calcul de 2^n présenté page 97 :

```
p = 1
while c != 0:
    p = p * 2
    c = c - 1
```

Mis à part le fait que l'on teste `c != 0` au lieu de `c > 0`, on peut penser que ce programme fonctionne comme le précédent calculant 2^n.

Par exemple, partant de l'état [c : 3], on obtient l'état final [c : 0 | p : 8].

Cependant, si on part de l'état [c : -1], on obtient successivement les états [c : -2 | p : 2], puis [c : -3 | p : 4], etc. Il est clair que *c* aura toujours une valeur différente de 0 puisqu'il s'agit d'un nombre négatif qui décroît. On dit que l'exécution est dans une boucle infinie, c'est-à-dire une boucle de laquelle il n'y a pas de possibilité de sortir.

Ce type d'erreur de conception est indétectable par l'ordinateur et il est assez fréquent, car les conditions et les corps des boucles sont en général plus complexes que dans cet exemple. Le seul moyen de sortir de la boucle infinie est d'interrompre volontairement l'exécution du programme, en pressant simultanément les touches *Ctrl* et *C* dans l'interpréteur.

Comme on l'a vu, il suffit de remplacer la condition par `c > 0` pour corriger le problème :

```
p = 1
while c > 0:
    p = p * 2
    c = c - 1
```

Savoir-faire Corriger un programme à l'aide d'un débogueur

1 On identifie les variables critiques, dont la valeur peut radicalement influencer le comportement du programme. En particulier, les variables intervenant dans la condition d'une instruction `while` sont particulièrement importantes puisqu'elles conditionnent le nombre de fois où le corps de cette boucle est exécuté.
2 On identifie, pour chacune de ces variables, les endroits clés du programme qui la concernent : par exemple, lorsqu'on lui affecte une valeur, en début ou en fin de boucle.
3 On crée un point d'arrêt à ces endroits, puis on déroule l'exécution du programme à l'aide du débogueur et on surveille les valeurs des variables critiques dans l'explorateur de variables.

Voir le chapitre 1 page 30 pour l'utilisation concrète du débogueur de Spyder.

Exercice 4.23 avec corrigé Utiliser un débogueur pour identifier l'erreur commise dans le programme ci-après. Le comportement attendu est le suivant : on demande des nombres entiers au clavier tant que la suite des nombres fournis est strictement croissante. La variable *n* compte le nombre de valeurs entrées au clavier.

```
precedent = int(input())
nouveau = int(input())
n = 2
while precedent < nouveau:
    nouveau = int(input())
    precedent = nouveau
    n = n + 1
```

Un premier test du programme permet de se rendre compte que celui-ci s'arrête dès le troisième nombre entré. Il n'est donc pas nécessaire de placer des points d'arrêt, on peut suivre l'évolution de `precedent` *et* `nouveau` *sur toute l'exécution du programme.*
On voit alors que le problème se pose aux deux premières lignes du corps de la boucle. L'instruction `nouveau = int(input())` *écrase la dernière valeur entrée au clavier, puis l'instruction* `precedent = nouveau` *rend forcément fausse la condition de la boucle.*
Il faut ici intervertir ces deux lignes pour que la variable `precedent` *mémorise correctement l'avant-dernière valeur tapée au clavier.*

Exercice 4.24 Le programme qui suit se termine-t-il pour toutes les valeurs initiales de *c* ?

```
p = 1
if c > 0:
    while c != 0:
        p = p * 2
        c = c - 2
```

Exercice 4.25 ** Pour accélérer le calcul de k^n, on se propose d'exploiter les identités suivantes, qui montrent comment calculer une puissance de k en remplaçant k par son carré et l'exposant par sa moitié :

$$\begin{cases} k^{2n} &= (k^2)^n \\ k^{2n+1} &= k(k^2)^n \end{cases}$$

L'algorithme est le suivant :

```
r = 1
while n > 0:
    if n % 2 == 1:
        r = r * k
    k = k**2
    n = n // 2
```

1 Quel est le rôle de la variable r dans cet algorithme ?
2 Établir la terminaison de cet algorithme.
3 Démontrer que cet algorithme est correct.
4 Cet algorithme est dit d'*exponentiation rapide*. Pour comprendre pourquoi, évaluer combien de multiplications il effectue et comparer avec la version naïve présentée plus haut dans ce chapitre.

4.4 Boucles inconditionnelles

On a pu voir dans la partie précédente que les boucles conditionnelles étaient nécessaires pour effectuer des calculs lorsqu'il n'est pas possible de borner le nombre d'étapes nécessaires. En pratique, dans de nombreux cas, on connaît à l'avance le nombre d'itérations qu'il faudra effectuer, ce qui rend inutile d'utiliser une boucle conditionnelle.

4.4.1 Boucle for

À l'aide de la construction `for c in range(n)`, on effectue un nombre d'itérations n donné. Ainsi, si on reprend le programme de calcul de 2^n, on aura juste à écrire :

```
p = 1
for c in range(n):
    p = 2 * p
```

Le compteur de boucle est ici entièrement géré par la boucle `for`. Il n'est pas besoin de l'incrémenter, ni de tester qu'il ne dépasse pas une certaine limite.

Il reste possible d'utiliser ce compteur au sein du corps de la boucle. Dans la boucle `for c in range(n)`, la variable `c` parcourt les entiers de 0 à $n-1$. On remarque que le paramètre passé à `range` est donc la valeur *avant* laquelle on s'arrête. Comme souvent en informatique, on numérote à partir de 0.

Ainsi, si l'on souhaite calculer la factorielle d'un entier n, définie par les relations :

$$0! = 1 \qquad (n+1)! = (n+1) \times n!$$

on écrit le programme suivant :

```
p = 1
for c in range(n):
    p = (c+1) * p
```

Exercice 4.26 Démontrer au moyen d'un invariant de boucle que le programme ci-dessus calcule bien $n!$ dans la variable *p*.

SAVOIR-FAIRE Écrire un programme utilisant une boucle `for`

1 On détermine combien de fois la boucle devra s'exécuter, ce nombre étant en général exprimé en fonction d'une ou plusieurs variable(s) du programme.
2 On choisit une variable pour le compteur et on identifie si elle doit jouer un rôle dans le corps de la boucle.
3 On écrit le corps de la boucle.
4 Comme pour la boucle `while`, il peut être nécessaire de prévoir une initialisation des variables en amont de la boucle et un post-traitement en aval.

Exercice 4.27 avec corrigé Écrire un programme qui calcule la n-ième puissance itérée de k, autrement dit le nombre $k^{k^{\cdot^{\cdot^{\cdot^{k}}}}}$ formé de n exemplaires de k.
Les exposants les plus hauts doivent être calculés en premier, sinon cette expression serait équivalente à $k^{k^{n-1}}$. Par exemple, la quatrième puissance itérée de 2 est $2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$ et non pas $\left(\left(2^2\right)^2\right)^2 = 2^{2\times 2\times 2} = 2^8 = 256$.

1 Puisque l'expression est formée de n exemplaires de k, il y a $n-1$ exponentiations successives à calculer, ce qu'on réalisera en autant d'itérations.

2 Ici, le compteur de boucle ne jouera aucun rôle dans le calcul, il peut prendre un nom générique comme i. La boucle commencera donc par :

```
for i in range(n-1):
```

3 Le corps de la boucle consiste à calculer une des exponentiations. Puisque les exposants sont à calculer « de haut en bas », on prévoit ici une variable r qui contient l'exposant déjà calculé au rang précédent, autrement dit la i-ème puissance itérée de k. Le corps de la boucle est donc simplement :

```
    r = k ** r
```

4 Il faut évidemment initialiser r avant d'entrer dans la boucle. Comme à la première itération, on veut calculer k^k, la valeur initiale correcte pour r est donc k. Le compteur i est entièrement géré par la boucle **for**, *il est inutile de l'initialiser.*

```
r = k
```

En sortie de boucle, r contient le résultat recherché, il n'y a donc aucun traitement supplémentaire.
On donne ci-après le programme complet obtenu.

```
r = k
for i in range(n-1):
    r = k ** r
```

En initialisant r à 1 et en effectuant n itérations, on a un programme équivalent et même correct pour $n = 0$. On ne pourra tester ce programme que pour de petites valeurs de n et k, car les nombres calculés croissent très rapidement. On obtient un résultat intéressant en prenant $k \simeq \sqrt{2}$.

SAVOIR-FAIRE Initialiser les variables

On a vu que, tant que l'on n'a pas affecté une valeur à une variable, celle-ci est absente de l'état. Il faut donc s'assurer qu'à chaque fois qu'une variable est utilisée dans une expression, elle est déjà présente dans l'état, sous peine d'une erreur. Pour cela :

1 On identifie la première ligne du programme où la variable est utilisée. Lorsque l'algorithme comporte des instructions conditionnelles, il peut y avoir plusieurs telles lignes pour la même variable, en fonction du résultat du test.

2 On vérifie que cette première ligne est toujours une affectation de cette variable. Elle peut être rendue difficile à repérer parce qu'une instruction **for** la gère ou parce qu'elle est combinée avec une saisie au clavier.

3 Enfin, si une variable n'est pas initialisée, il faut déterminer une valeur d'initialisation cohérente avec la suite du programme et ajouter l'instruction correspondante avant la première utilisation de la variable.

Exercice 4.28 avec corrigé Corriger le programme suivant.

```
n = int(input())
for i in range(n):
    if i % 2 == 0:
        carre = i ** 2
    else:
        carre = 0
    total = total + carre
```

Ce programme semble prévu pour calculer la somme des carrés des entiers pairs strictement inférieurs à n. Il comporte quatre variables :

- `n` *est initialisée par une entrée au clavier en tout début de programme.*
- `i` *est un compteur de boucle et n'a donc pas besoin d'être initialisée.*
- `carre` *apparaît pour la première fois dans l'instruction conditionnelle et elle y est correctement initialisée, quel que soit le résultat du test* `i % 2 == 0`.
- `total` *n'apparaît que dans la dernière ligne de la boucle. Cette ligne est bien une affectation de* `total`*, mais avec une expression qui dépend elle-même de* `total` *: il sera impossible d'évaluer cette expression lors de la première itération. Il faut donc initialiser* `total` *avant d'entrer dans la boucle, avec la valeur 0 pour ne pas perturber la somme qui sera calculée par la suite.*

4.4.2 Valeurs itérables

Il y a un autre cas de figure qui nécessite de pouvoir faire une boucle : lorsque l'on veut manipuler des valeurs de type composé, c'est-à-dire regroupant plusieurs éléments, tels les n-uplets, les chaînes de caractères ou les listes. On aimerait pouvoir effectuer une opération sur chacun des éléments d'une de ces valeurs, ce qui est raisonnable car :

- on sait identifier le premier élément que l'on va traiter (c'est le premier caractère d'une chaîne, la première composante d'un n-uplet...) ;
- si on vient de traiter un élément, on sait que le suivant est juste après.

Toute valeur de type composé sur laquelle ces deux opérations sont possibles est dite *itérable*.

Savoir identifier le dernier élément à traiter est également utile pour assurer la terminaison de la boucle, mais pas indispensable à proprement parler pour effectuer l'itération.

Pour construire une boucle qui parcourt un itérable, on utilise la syntaxe suivante :

```
for element in iterable:
    corps_de_boucle
```

Par exemple, voici comment calculer la somme des éléments d'un n-uplet :

```
t = (1, 5, 9)
s = 0
for e in t:
    s = s + e
```

De même, voici comment compter les espaces dans une chaîne de caractères :

```
chaine = 'Les itérables sont vraiment épatants.'
esp = 0
for c in chaine:
    if c == ' ':
        esp += 1
```

4.4.3 L'itérable range

L'expression `range(n)` est en fait aussi un itérable, défini de la façon suivante :

- Son premier élément est 0.
- Son plus grand élément est $n - 1$.
- Après l'élément i vient l'élément $i + 1$.

Cette expression admet d'ailleurs deux formes plus générales, `range(m,n)` (le premier élément est m et le dernier est n) et `range(m,n,p)` (l'élément venant après i est $i + p$).

Si on évalue une expression `range` dans un interpréteur interactif, on obtient la réponse

```
In [6]: range(5)
Out[6]: range(0, 5)
```

car Python ne calcule pas explicitement cette séquence d'éléments. On peut, en revanche, récupérer une liste par conversion explicite :

```
In [7]: list(range(5))
Out[7]: [0, 1, 2, 3, 4]
```

En pratique Dans les versions Python 2.x

Ce comportement est différent : `range` renvoie toujours une liste. L'avantage de ne pas calculer cette plage de valeurs en Python 3.x est qu'on peut ainsi utiliser des valeurs comme `range(2**32)` qui seraient très coûteuses à calculer.

Savoir-faire Choisir entre une boucle `for` et une boucle `while`

Si on connaît à l'avance le nombre de répétitions à effectuer, ou plus généralement, si on veut parcourir une valeur itérable, on choisit une boucle `for`.

À l'inverse, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle `while` qu'il faut choisir.

Exercice 4.29 avec corrigé Quelle boucle est adaptée à l'écriture de programmes traitant les problèmes suivants :

1 Calculer la valeur absolue d'un nombre donné.
2 Calculer la norme d'un vecteur dans un espace vectoriel de dimension finie (pour une des normes usuelles).
3 Déterminer tous les diviseurs d'un nombre entier donné.
4 Déterminer le plus petit diviseur (différent de 1) d'un nombre entier donné.
5 Déterminer la date du prochain vendredi 13 connaissant la date d'aujourd'hui.

1 *Il n'y a pas besoin de boucle.*
2 *Une boucle* `for` *: le nombre d'itérations sera la dimension de l'espace vectoriel car toutes les normes usuelles demandent de parcourir les coordonnées du vecteur. Il est d'ailleurs probable qu'on représente un vecteur sous forme d'une valeur itérable.*
3 *Une boucle* `for` *: tous les diviseurs de n sont compris entre 1 et n. Si on s'y prend bien, on peut obtenir l'ensemble des diviseurs de n en parcourant un intervalle nettement moins grand, mais même avec cet algorithme amélioré, la boucle* `for` *reste la plus adaptée.*
4 *Une boucle* `while` *: il n'est pas nécessaire de parcourir tous les diviseurs de n, on s'arrête au premier rencontré, ce que l'on traduit dans la condition de la boucle.*
5 *En principe, une boucle* `while` *: on ne sait pas dans combien de mois aura lieu le prochain vendredi 13 puisque c'est justement ce qu'on cherche. Il est certainement possible de trouver une formule qui répond directement à la question, mais cela demande un travail préliminaire à l'écriture du programme.*

Exercice 4.30 Écrire les programmes proposés dans l'exercice précédent.

4.4.4 Interrompre une boucle

Dans la section précédente, on a vu que selon ce qu'on cherche à programmer, l'une des deux boucles est plus appropriée que l'autre. Pour certains problèmes, la décision n'est pas si simple.

On considère par exemple un programme qui détermine si l'entier n est premier : il faut *a priori* parcourir les entiers entre 2 et $\frac{n}{2}$ et vérifier qu'aucun d'entre eux ne divise n.

En réalité, on peut s'arrêter à $\sqrt{n}$ car si $n = p \times q$ avec $p > \sqrt{n}$ alors q est un diviseur de n inférieur à $\sqrt{n}$. On peut donc écrire le programme suivant :

```
import math
premier = True
for i in range(2, int(math.sqrt(n))+1):
    if n % i == 0:
        premier = False
```

La racine carrée est calculée à l'aide de la fonction `math.sqrt`. On n'oubliera pas l'instruction `import math` pour y avoir accès.

Cependant, on s'aperçoit vite que lorsque n n'est pas premier, ce programme effectue des calculs inutiles, puisqu'on pourrait arrêter la boucle dès qu'on a trouvé un diviseur. Une première solution consiste à réécrire la boucle `for` en une boucle `while`, ce qui est toujours possible. On peut alors préciser dans la condition du `while` qu'il faut s'arrêter dès qu'un diviseur est trouvé, autrement dit dès que *premier* prend la valeur `False`.

```
premier = True
i = 2
while i <= int(math.sqrt(n)) and premier:
    if n % i == 0:
        premier = False
    i = i + 1
```

Une autre approche est de conserver la boucle `for` et d'en sortir au moyen de l'instruction `break` dès qu'on rencontre un diviseur.

```
premier = True
for i in range(2, int(math.sqrt(n))+1):
    if n % i == 0:
        premier = False
        break
```

Ainsi, il n'est pas nécessaire de gérer le compteur de boucle à la main. Cette méthode reste à utiliser avec parcimonie, car dans des programmes plus conséquents, elle peut compliquer la compréhension des différents cas de sortie de la boucle.

4.4.5 Boucles imbriquées

Quand l'instruction à exécuter à l'intérieur d'une boucle est elle aussi répétitive, le corps de cette boucle contient une seconde boucle et on dit qu'elles sont imbriquées. Les bornes de la boucle interne dépendent souvent du compteur de la boucle externe.

SAVOIR-FAIRE Construire des boucles imbriquées

Imbriquer des boucles est judicieux si on a identifié une procédure « doublement répétitive ».

1 Si l'une des instructions qui se répète dépend de l'autre, il faut qu'elle constitue la boucle interne.
2 De même, si l'une des instructions doit se répéter sans « interférence » entre ses itérations, elle doit être la boucle interne.
3 Ensuite, on s'interroge pour savoir si le compteur de la boucle interne doit ou non dépendre de celui de la boucle externe.
4 On écrit le corps la boucle interne.
5 On écrit enfin le corps de la boucle externe, ce qui se réduit souvent à ajouter quelques instructions avant ou après le corps de la boucle interne.
6 On prendra garde aux trois niveaux d'indentation présents dans un tel programme.

Exercice 4.31 avec corrigé Écrire un programme qui affiche à l'écran un rectangle de n lignes sur p colonnes rempli de caractères * sur le modèle suivant (pour $n = 3$ et $p = 5$) :

```
*****
*****
*****
```

Modifier ensuite ce programme pour qu'il affiche un triangle rectangle isocèle de taille n constitué de caractères * sur le modèle suivant (pour $n = 3$) :

```
*
**
***
```

Il faut écrire n lignes identiques, mais nous ne connaissons pour l'instant aucun moyen de créer directement une chaîne de p caractères identiques. Nous allons donc utiliser des boucles imbriquées.

Une fois que l'on est passé à la ligne, il est impossible de remonter. Il faut donc que la boucle interne affiche une ligne complète et que la boucle externe reproduise n lignes.

Les lignes étant identiques, la boucle interne ne dépend pas de la boucle externe.

La boucle interne consiste simplement à afficher p étoiles sans revenir à la ligne.

```
for j in range(p):
    print("*", end="")
```

La boucle externe se contente de répéter n fois la boucle externe et de revenir à la ligne à la fin de chaque itération. Le programme complet est donné ci-après.

```
for i in range(n):
    for j in range(p):
        print("*", end="")
    print()
```

Dans le second programme demandé, toutes les lignes ne sont pas identiques : la boucle interne dépend donc de la boucle externe. Il est facile de voir qu'il faudra afficher i étoiles sur la i-ème ligne. Il suffit donc de modifier le nombre d'itérations de la boucle interne, en prenant garde que i va parcourir les valeurs de 0 à $n - 1$.

```
for i in range(n):
    for j in range(i+1):
        print("*", end="")
    print()
```

Exercice 4.32 Écrire des programmes qui affichent les triangles de taille n constitués de caractères * sur les modèles suivants (pour $n = 3$) :

1
```
  *
 **
***
```
2
```
***
 **
  *
```
3
```
  *
 ***
*****
```

Exercice 4.33 * Écrire un programme de construction du tableau périodique des éléments au moyen de boucles imbriquées.

4.5 Exercices

Exercice 4.34 Reprendre le programme de l'exercice 4.8 et l'adapter pour qu'en cas d'excès de vitesse, il donne le montant de la contravention encourue. Celle-ci est de 68 € si l'excès commis est inférieur à 20 km/h, 135 € au delà.

Exercice 4.35 Écrire un programme qui détermine l'ordre de grandeur d'un nombre x non nul donné, autrement dit l'entier relatif n tel que $10^n \leqslant |x| < 10^{n+1}$. On n'utilisera pas les fonctions logarithmes du module **math**.

Exercice 4.36 Écrire un programme qui trouve le plus petit multiple commun à 2 entiers naturels m et n.

Exercice 4.37

1 Écrire un programme qui reçoit comme donnée un quadruplet $(jours, heures, minutes, secondes)$ et vérifie que celui-ci respecte les conventions habituelles sur les durées (secondes entre 0 et 59, etc.).
2 Écrire un programme qui reçoit une durée exprimée en secondes et construit un quadruplet $(jours, heures, minutes, secondes)$ désignant la même durée et respectant les conventions habituelles sur les durées.

Exercice 4.38 *

1 Écrire un programme qui calcule la valeur d'un entier connaissant son écriture binaire fournie sous forme d'une chaîne de caractères.
2 Écrire un programme qui construit dans une chaîne de caractères l'écriture binaire d'un entier naturel donné.
3 Généraliser à une base quelconque. Pour des raisons pratiques, on pourra se limiter aux bases inférieures ou égales à 36 ; voyez-vous pourquoi ?

5

Fonctions

Dans ce chapitre, nous verrons qu'on peut définir une fonction pour isoler une instruction qui revient plusieurs fois dans un programme. Une fonction est définie par :

- *son nom ;*
- *ses arguments qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel ;*
- *éventuellement une valeur de retour communiquée au programme par la fonction en fin d'exécution.*

On souhaite écrire un programme qui affiche de la manière suivante les horaires de trois trains pour Brest, Strasbourg et Toulouse :

```
Le train en direction de Brest partira à 9h
-------------------------------------------------------
Le train en direction de Strasbourg partira à 9h30
-------------------------------------------------------
Le train en direction de Toulouse partira à 9h45
-------------------------------------------------------
```

On peut écrire pour cela le programme suivant :

```
print("Le train en direction de", "Brest", "partira à", "9h")
print("-------------------------------------------------------")
print("Le train en direction de", "Strasbourg", "partira à", "9h30")
print("-------------------------------------------------------")
print("Le train en direction de", "Toulouse", "partira à", "9h45")
print("-------------------------------------------------------")
```

Dans ce programme, on fait trois fois la même chose : on affiche la chaîne `"Le train en direction de"`, puis la destination, puis la chaîne `"partira à"`, puis l'horaire et enfin la ligne de séparation. Répéter plusieurs fois la même chose dans un programme n'est pas souhaitable. En particulier, si on procède par *copier-coller*, alors toute erreur sera dupliquée autant de fois. Si par exemple on fait une faute d'orthographe à `"partira"`, alors il faudra la corriger trois fois. De même, si on souhaite modifier le comportement du programme, par exemple pour afficher des étoiles à la place des tirets, alors il faudra effectuer la modification à trois endroits dans le code. Ici, une solution pour éviter la duplication consiste à utiliser une boucle `for` qui parcourt une liste de paires destination-horaire :

```
for (d, h) in [("Brest", "9h"), ("Strasbourg", "9h30"), ("Toulouse", "9h45")]:
    print("Le train en direction de", d, "partira à", h)
    print("-------------------------------------------------------")
```

Ainsi, pour chaque couple d'une destination `d` et d'un horaire `h`, on effectue les deux instructions suivantes :

```
    print("Le train en direction de", d, "partira à", h)
    print("-------------------------------------------------------")
```

Cela ne suffit pas nécessairement à éliminer toute redondance. En effet, on peut imaginer avoir besoin d'afficher un départ de train à un autre endroit du programme. Il faudrait alors copier les deux instructions, avec les valeurs de `d` et `h` pertinentes. Ce qu'il nous faut donc, c'est un moyen d'*isoler* ces deux instructions. C'est exactement ce que permet de faire une *fonction*.

5.1 La notion de fonction

Une fonction est une suite d'instructions qui dépend de paramètres. Ainsi, dans l'exemple précédent, on peut définir une fonction appelée **annoncer_train** et paramétrée par une destination *d* et un horaire *h* de la façon suivante :

```
def annoncer_train(d, h):
    print("Le train en direction de", d, "partira à", h)
    print("------------------------------------------------------------")
```

Comme on le voit sur cet exemple, une fonction est introduite par le mot-clé **def**, suivi du nom de la fonction, de ses paramètres entre parenthèses, de deux-points, puis d'un bloc d'instructions. Ce dernier s'appelle le *corps* de la fonction. Il doit donc être indenté et la fin de ce bloc marque la fin de la définition de fonction. Les variables *d* et *h*, qui figurent comme paramètres dans la définition de la fonction, s'appellent les *arguments formels*.

En utilisant la fonction **annoncer_train**, le programme se réduit alors aux trois instructions suivantes :

```
annoncer_train("Brest", "9h")
annoncer_train("Strasbourg", "9h30")
annoncer_train("Toulouse", "9h45")
```

Chaque instruction **annoncer_train(...)** est un *appel* à la fonction. Dans l'appel **annoncer_train("Brest", "9h")**, les expressions "Brest" et "9h", que l'on donne en arguments, s'appellent les arguments *effectifs* de l'appel.

Organiser un programme à l'aide de fonctions évite les redondances. En outre, cela les rend plus clairs et plus faciles à lire : pour comprendre le programme précédent, il n'est pas nécessaire de savoir comment la fonction **annoncer_train** est programmée, il suffit de savoir ce qu'elle fait. Enfin, cela permet d'organiser l'écriture du programme. On peut décider d'écrire la fonction un jour et le programme principal le lendemain. On peut aussi organiser une équipe de manière à ce qu'un programmeur écrive la fonction et un autre le programme principal.

Ce mécanisme peut se comparer à celui des définitions du langage mathématique qui permet d'utiliser le mot « groupe » au lieu de répéter la locution « ensemble muni d'une loi interne, associative, ayant un élément neutre, dans lequel tout élément a un symétrique ». Pour poursuivre la comparaison, le langage mathématique utilise aussi de telles définitions paramétrées : « Une fonction de classe $\mathcal{C}^n$ est... », « Un $\mathbb{K}$-espace vectoriel est... », etc.

5.1.1 Le retour de valeur

Certaines fonctions sont capables d'effectuer un *calcul*, et non pas seulement un affichage comme dans **annoncer_train**. Par exemple, on peut souhaiter écrire une fonction **hypotenuse**, avec deux paramètres x et y, pour calculer $\sqrt{x^2 + y^2}$. En Python, on la définit de la manière suivante :

```
def hypotenuse(x, y):
    z = math.sqrt(x*x + y*y)
    return z
```

On l'utilise, par exemple pour calculer $\sqrt{3^2 + 4^2}$, de la manière suivante :

```
In [1]: hypotenuse(3.0, 4.0)
Out[1]: 5.0
```

Dans cette instruction, l'appel `hypotenuse(3.0, 4.0)` transmet les arguments effectifs `3.0` et `4.0` à la fonction, en lieu et place des arguments formels *x* et *y*. La fonction `hypotenuse` calcule alors la valeur `math.sqrt(x*x + y*y)` et la renvoie. En Python, ce retour de valeur est effectué par l'instruction `return`. C'est le processus inverse du passage d'arguments, qui transmet des informations du programme principal vers le corps de la fonction. L'appel à la fonction `hypotenuse` constitue alors une expression, dont la valeur est la valeur renvoyée, à savoir `5.0`. Celle-ci est affichée par l'instruction `print`.

Si on oublie de faire précéder l'expression renvoyée par le mot-clé `return`, comme dans la fonction `hypotenuse` suivante :

```
def hypotenuse(x, y):
    math.sqrt(x*x + y*y)
```

celle-ci est toujours acceptée, mais elle ne renvoie pas de valeur. Ainsi, si on affiche le résultat du calcul de cette fonction

```
In [2]: hypotenuse(3.0, 4.0)
Out[2]: None
```

on obtient, non pas la valeur attendue `5.0`, mais une valeur particulière `None`. De manière générale, l'absence d'instruction `return` dans une fonction se traduit par une instruction implicite `return None`. C'était en particulier le cas pour la fonction `annoncer_train` précédente. Par opposition aux fonctions qui renvoient des valeurs, on pourra en général considérer l'appel à une fonction sans `return` (ou qui renvoie `None`) comme une instruction. On verra cependant dans la prochaine section que certaines fonctions sont à la fois des expressions et des instructions.

ATTENTION Ne pas confondre `return` et `print`

Il arrive que l'on confonde ces deux intructions à cause de l'interpréteur interactif, qui affiche la valeur renvoyée par `f` lorsqu'on y saisit `f(...)`. Leurs rôles sont en réalité totalement différents :

- L'instruction `print` n'a pas de valeur, elle a pour seul effet d'afficher un texte à l'écran.
- L'instruction `return`, au contraire, n'affiche rien mais décide de ce que renvoie la fonction, et donc de la valeur de l'appel `f(...)`.

Ainsi, si on commet l'erreur d'utiliser `print` à la place de `return` dans une fonction, celle-ci affichera la valeur calculée à l'écran mais ne la renverra pas (elle renverra `None` comme on l'a vu plus haut).

De manière générale, il est important dans un algorithme de distinguer le résultat calculé des sorties affichées à l'écran. De même, il faut distinguer les données sur lesquelles on travaille des entrées tapées au clavier.

Quand elle ne se trouve pas à la fin d'une fonction, mais au milieu, l'instruction `return` a pour effet d'interrompre le déroulement de la fonction. Ainsi, on peut écrire :

```
def puissance(x, n):
    if (x == 0):
        return 0
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Si la valeur de `x` est nulle, l'instruction `return 0` interrompra le déroulement de la fonction et aucune des instructions suivantes (à partir de `r = 1`) ne sera exécutée.

SAVOIR-FAIRE **Concevoir une fonction**

Quand on conçoit une fonction, il est préférable de lui donner un nom explicite, car elle est susceptible d'apparaître à de nombreux endroits dans le programme (On n'imagine pas que les fonctions de la bibliothèque Python s'appellent `f1`, `f2`, etc.). En revanche, les noms des arguments formels peuvent être courts, car leur portée, et donc leur signification, est limitée au corps de la fonction.

Il convient par ailleurs de documenter convenablement les fonctions, en *spécifiant* les hypothèses faites sur les arguments formels, leur relation avec le résultat renvoyé, mais aussi les effets de la fonction (affichage, etc.) le cas échéant. Python propose un mécanisme pour associer une documentation à toute fonction, sous la forme d'une chaîne de caractères placée au début du corps de la fonction. Ainsi, on peut écrire :

```
def puissance(x, n):
    """calcule x à la puissance n, en supposant x > 0 et n >= 0"""
    r = 1
    for i in range(n):
        r = r * x
    return r
```

La documentation d'une fonction `f` peut être affichée avec `help(f)`.

Exercice 5.1 Écrire une fonction qui prend en arguments deux entiers x et y et qui renvoie -1 si $x < y$, 0 si $x = y$ et 1 si $x > y$.

Exercice 5.2 avec corrigé Écrire une fonction qui prend en arguments deux entiers $n \geqslant 0$ et $d > 0$ et qui renvoie un couple formé du quotient et du reste de la division euclidienne de n par d. Le quotient et le reste seront calculés par soustractions successives.

Le nom à donner à la fonction, à ses arguments et à ses variables sont pour ainsi dire donnés dans l'énoncé. On n'oublie pas de mentionner les conditions d'utilisation dans la documentation.

```
def division_euclidienne(n, d):
    """calcule q et r tels que n = q * d + r et 0 <= r < d,
    en supposant n >= 0 et d > 0
    """
    r = n
    q = 0
    while r > d:
        r = r - d
        q = q + 1
    return (q, r)
```

SAVOIR-FAIRE Utiliser un débogueur avec des fonctions

On a vu jusqu'ici que, en mode débogueur, on pouvait exécuter un programme instruction par instruction à l'aide de la commande *Pas en avant* (ou `next`). Lorsque le programme comporte des fonctions, cette commande considère chaque appel de fonction comme une instruction unique et passe donc directement à l'instruction suivante. Ce n'est pas approprié si on veut voir ce qui se passe dans le corps de la fonction.

Pour entrer dans le corps d'une fonction au moment où elle est appelée, on clique sur *Pas vers l'intérieur* ou on tape `s` comme `step`. Le curseur qui indique la prochaine ligne à exécuter se déplace sur la première ligne du corps de la fonction et on peut ensuite suivre son déroulement instruction par instruction. Les variables utilisées dans le corps de la fonction s'affichent alors également dans l'explorateur de variables.

À l'inverse, si le débogueur est entré dans une fonction et si on souhaite aller directement à la fin de l'exécution de cette fonction, au moment du retour de valeur, on clique sur *Pas vers l'extérieur* ou on tape `r` comme `return`.

Exercice 5.3 avec corrigé On donne le programme suivant :

```
def f(x):
    for i in range(100):
        x = (13 * x + 1) % 256
    return x

def g():
    s = 0.
    for j in range(10):
        a = f(j)
        s = s + 1. / (a-210)
    return s

print(g())
```

1 Que se passe-t-il lors de son exécution ?
2 Jusqu'à quel point la simple lecture du code permet-elle d'expliquer ce comportement ?
3 À quelle itération de la boucle de la fonction **g** se situe le problème ?

1 Le programme s'arrête sur l'erreur suivante :

```
ZeroDivisionError: 'float division by zero'
```

2 Il est clair que la division par zéro provient de la ligne `s = s + 1. / (a-210)` *dans la fonction* **g***; en revanche, la lecture seule du code ne nous permet pas de déterminer quand la variable* `a` *prend la valeur* 210.

3 Si l'on n'utilise que la commande Pas en avant*, le débogueur n'entre pas dans l'appel à la fonction* **g** *et on ne voit pas ce qui se passe.*
À la ligne `print(g())`*, il faut donc utiliser* Pas vers l'intérieur*. Ensuite, on peut effectuer* Pas en avant *dans le corps de* **g** *jusqu'à déterminer que la division par zéro a lieu à l'itération* `j=6`*. Si par erreur on continue à utiliser* Pas vers l'intérieur*, on peut se retrouver dans le corps de* **f***. Pour ne pas avoir à exécuter les* 100 *itérations à la main, on peut effectuer* Pas vers l'extérieur*.*
Remarquons que placer un point d'arrêt à la ligne qui produit la division par zéro évite de dérouler toutes les itérations dans **g** *à la main.*

POUR ALLER PLUS LOIN Un générateur pseudo-aléatoire

Le lecteur qui aura eu la curiosité de dérouler une exécution de la fonction `f` dans l'exercice corrigé ci-avant aura pu remarquer le caractère apparemment imprévisible des valeurs successives prises par la variable *x* (apparemment seulement puisqu'il s'agit d'une suite périodique). Ce sont des procédures telles que celle-ci qui permettent aux ordinateurs de simuler le hasard. On parle de générateur *pseudo-aléatoire*.

5.1.2 Variables globales et locales

En Python, on distingue deux sortes de variables : les *globales* et les *locales*. Par exemple, dans le programme suivant, *x* est une variable globale :

```
x = 7
print(x)
```

À l'inverse, la variable *y* dans la fonction `f` suivante est locale :

```
def f():
    y = 8
    return y
```

Ainsi, avant un appel à `f()`, on se trouve dans l'état :

x
7

Pendant l'appel, et après avoir exécuté `y = 8`, on se trouve dans un état augmenté d'une variable locale *y*, c'est-à-dire :

x	y
7	8

Après l'appel, la variable locale *y* disparaît et on se retrouve dans l'état initial :

x
7

En particulier, l'instruction suivante échoue en indiquant que la variable *y* n'est pas définie :

```
print(y)
```

On dit que la *portée* de la variable *y* est limitée au corps de la fonction `f`. Les variables globales, elles, ont une portée qui s'étend généralement sur l'ensemble du programme.

Si on souhaite faire référence à une variable globale dans une fonction, par exemple pour écrire une fonction qui réinitialise la variable globale *x* à 0, alors il ne faut pas écrire :

```
def reinitialise():
    x = 0
```

En effet, cela ne fait qu'affecter une variable locale *x* à la fonction `reinitialise`. On peut s'en convaincre en exécutant le code suivant :

```
reinitialise()
print(x)
```

et en observant qu'il affiche toujours la valeur `7`. Pour que la fonction `reinitialise` puisse avoir accès à la variable globale *x*, il faut désigner cette dernière comme telle :

```
def reinitialise():
    global x
    x = 0
```

Ainsi, le programme :

```
reinitialise()
print(x)
```

affiche maintenant `0` et non `7`.

De manière générale, si elle n'y est pas explicitement déclarée comme globale, une variable *x* est locale à la fonction dans le corps de laquelle elle est affectée. Elle est globale si elle est utilisée dans la fonction sans être affectée ou si elle est déclarée globale.

```
def f():
    global a
    a = a + 1
    c = 2 * a
    return a + b + c
```

Dans cette fonction, *a* est globale car elle est déclarée comme telle, *b* est globale car elle est utilisée mais non affectée et *c* est locale car elle est affectée mais n'est pas déclarée globale.

Si une variable *x* est déclarée globale dans une fonction `f` mais pas dans une fonction `g`, et si le nom de variable *x* est utilisé dans `g`, comme dans l'exemple suivant :

```
def f():
    global x
    x = 2

def g():
    x = 3
```

alors il faut considérer qu'on a deux variables *x* différentes : une globale et une locale dans **g**. Dans la fonction **g**, le nom *x* désigne la variable locale et non la globale, qui du fait de son homonymie, ne peut être utilisée. C'est le seul cas où une variable globale a une portée qui ne couvre pas l'ensemble du programme. Ainsi, le programme suivant affiche `2` car l'affectation `x = 3` concerne la variable *x* locale à **g** et non la variable globale *x* :

```
x = 1
f()
g()
print(x)
```

Ainsi, un appel à une fonction peut modifier l'état du programme principal. C'est évidemment possible aussi avec une fonction qui renvoie une valeur : un appel à une telle fonction est donc à la fois une expression (puisqu'il prend une valeur) et une instruction (puisqu'il modifie l'état).

SAVOIR-FAIRE Du bon usage des variables globales

De façon générale, une bonne pratique consiste à utiliser les variables globales pour représenter les *constantes* du problème. En pratique, on ne devrait pas recourir souvent à la construction **global** de Python.

Comme pour les fonctions, il est préférable de donner aux variables globales des noms longs et explicites, ce qui les distinguera de fait des variables locales qui portent habituellement des noms courts (comme les paramètres formels).

Exercice 5.4 Quelles sont les variables locales et globales de la fonction **f** ? Qu'affiche le programme suivant ?

```
def f():
    global a
    a = a + 1
    c = 2 * a
    return a + b + c

a = 3
b = 4
c = 5
print(f())
print(a)
print(b)
print(c)
```

5.1.3 Ordre d'évaluation

Comme les expressions peuvent modifier la mémoire, l'ordre dans lequel les arguments d'une fonction sont évalués peut avoir une influence sur le résultat. Ainsi, le programme

suivant affiche le résultat 2 si les arguments de `somme` sont évalués de gauche à droite et 3 sinon :

```
n = 0
def g(x):
    global n
    n = n + 1
    return x + n
def somme(x, y):
    return x + y
print(somme(n, g(1)))
```

Il se trouve qu'en Python, l'ordre d'évaluation est toujours de la gauche vers la droite. Ce n'est pas le cas dans d'autres langages de programmation, où il peut être au contraire de la droite vers la gauche, voire non spécifié. Même en ne considérant que Python, rien ne garantit que l'ordre d'évaluation ne changera pas dans de prochaines versions. Aussi est-il important de ne jamais écrire un programme dont le résultat en dépende. Dans l'exemple précédent, on peut par exemple forcer `g(1)` à être évaluée en premier, en stockant sa valeur dans une variable *a* :

```
a = g(1)
print(somme(n, a))
```

Exercice 5.5 Donner un exemple de programme montrant que l'évaluation des arguments de l'opérateur + se fait également de gauche à droite.

Exercice 5.6 Qu'affiche le programme suivant ? Pourquoi ?

```
def f(x):
    global b
    b = 10
    return x + 1

a = 3
b = 4
print(f(a)+b)
```

Proposer une adaptation de ce programme dans laquelle le résultat affiché ne dépend pas de l'ordre d'évaluation.

5.1.4 Passage par valeur

On considère la fonction `f` suivante, qui incrémente son argument *x* et en renvoie ensuite la valeur :

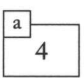
```
def f(x):
    x = x + 1
    return x
```

Alors le programme :

```
a = 4
print(f(a))
```

affiche sans surprise la valeur 5.

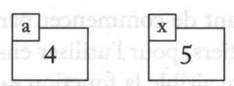
De manière plus surprenante, la variable `a` contient toujours la valeur `4` après l'appel à `print(f(a))`. En effet, on part de l'état suivant :



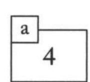
Immédiatement après l'appel à `f(a)`, on se retrouve dans un état :



Une nouvelle variable `x` a reçu la *valeur* de `a`. Après l'instruction `x = x + 1`, on se retrouve dans l'état :



Seule `x` a été modifiée. La fonction `f` renvoie alors la valeur de `x`, soit `5`, puis `x` disparaît et on se retrouve dans l'état de départ :



De ce point de vue, il n'y a pas de différence entre un paramètre formel et une variable locale : les deux ont une durée de vie limitée à l'appel de la fonction.

Ce mécanisme de transmission des arguments s'appelle le *passage par valeur*, car c'est seulement la valeur de l'argument effectif qui est transmise à la fonction appelée, et non l'argument effectif lui-même.

ATTENTION Liaison dynamique en Python

On considère le programme suivant qui définit une fonction `f` appelée depuis une fonction `g` :

```
def f():
    print('fonction 1')
def g():
    f()
```

Sans surprise, le message `'fonction 1'` s'affiche si on appelle `g()`. Rien n'empêche de redéfinir ensuite la fonction `f`, par exemple pour qu'elle affiche plutôt `'fonction 2'` :

```
def f():
    print('fonction 2')
```

De manière plus surprenante, c'est maintenant le message `'fonction 2'` qui est affiché quand on appelle de nouveau `g()`, alors qu'il s'agit toujours de la *même* fonction `g`. On appelle ce phénomène la *liaison dynamique*. Dit simplement, au moment de l'appel à une fonction `f`, Python en considère la dernière définition.

D'autres langages de programmation proposent au contraire une liaison *statique* où, dans l'exemple précédent, c'est toujours la fonction `f` initiale qui est appelée par la fonction `g` et donc toujours `'fonction 1'` qui est affiché.

Exercice 5.7 Expliquer pourquoi il n'est pas possible d'écrire une fonction **echange** qui échange le contenu des deux variables entières passées en arguments ?

5.2 Mécanismes avancés

5.2.1 Fonctions locales

Il arrive que l'utilisation d'une fonction soit limitée à la définition d'une autre fonction. On va supposer, par exemple, qu'on veut écrire une fonction **max3** calculant le maximum de trois entiers. Pour cela, il est élégant de commencer par la définition d'une fonction **max2** calculant le maximum de deux entiers, pour l'utiliser ensuite deux fois. Cependant, on ne souhaite pas nécessairement rendre visible la fonction **max2**. On la définit donc localement à la fonction **max3**, de la manière suivante :

```
def max3(x, y, z):
    def max2(u, v):
        if u > v:
            return u
        else:
            return v
    return max2(x, max2(y, z))
```

On parle alors de *fonction locale* à une autre fonction.

Plus subtilement encore, une fonction locale peut faire référence à des arguments, ou à des variables locales, de sa fonction englobante :

```
def f(x, y, z):
    a = x * x
    def g(n):
        return n + a
    return g(y) + g(z)
```

5.2.2 Fonctions comme valeurs de première classe

En Python, une fonction est une valeur comme une autre, c'est-à-dire qu'elle peut être passée en argument, renvoyée comme résultat ou encore stockée dans une variable. On dit que les fonctions sont des *valeurs de première classe*.

Une application directe est la définition d'un opérateur mathématique par une fonction. Par exemple, la somme $\sum_{i=0}^{n} f(i)$, pour une fonction f quelconque, peut être ainsi définie :

```
def somme_fonction(f, n):
    s = 0
    for i in range(n+1):
        s = s + f(i)
    return s
```

Pour calculer la somme des carrés des entiers de 1 à 10, on commence par définir une fonction `carre`.

```
def carre(x):
    return x*x
```

Puis, on la passe en argument à la fonction `somme_fonction` :

```
In [3]: somme_fonction(carre, 10)
Out[3]: 385
```

On peut même éviter de nommer la fonction `carre`, puisqu'elle est réduite à une simple expression, en utilisant une *fonction anonyme*. Une telle fonction s'écrit `lambda x: e`, où `e` est une expression pouvant comporter la variable `x`. Elle désigne la fonction $x \mapsto e(x)$. Ainsi l'exemple précédent se réécrit-il plus simplement :

```
In [4]: somme_fonction(lambda x: x*x, 10)
Out[4]: 385
```

De même qu'on peut passer une fonction en argument, on peut renvoyer une fonction comme résultat. En particulier, il est possible d'écrire une fonction qui prend comme arguments deux fonctions f et g et qui renvoie la composée $f \circ g$:

```
def composee(f, g):
    def h(x):
        return f(g(x))
    return h
```

Là encore, on simplifie l'écriture en utilisant une fonction anonyme :

```
def composee(f, g):
    return lambda x: f(g(x))
```

Il est maintenant possible de composer la fonction `carre` avec elle-même et de stocker le résultat dans une variable `a` :

```
In [5]: a = composee(carre, carre)
```

Ensuite, on applique la fonction contenue dans `a` à un argument :

```
In [6]: a(4)
Out[6]: 256
```

Les fonctions de première classe servent à exprimer de façon élégante des algorithmes génériques qu'on serait sinon obligé de réécrire pour chaque fonction. Des exemples d'application en sont donnés dans les chapitres 8 et 9.

Exercice 5.8 * Écrire une fonction `derive` qui prend en argument une fonction f et un flottant $\epsilon > 0$ et renvoie la fonction :

$$x \mapsto \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Vérifier que la fonction renvoyée par `derive(carre, 0.001)` est effectivement une approximation de $x \mapsto 2x$.

5.2.3 Fonctions partielles

Les fonctions considérées jusqu'à présent sont des *fonctions totales*, c'est-à-dire qu'elles renvoient toujours un résultat, quelle que soit la valeur de leurs arguments. Il existe aussi des fonctions dites *partielles* parce qu'elles ne renvoient pas un résultat pour *toutes* les valeurs possibles des arguments. Par exemple, la fonction **divise** ci-après est partielle :

```
def divise(x, y):
    return x // y
```

En effet, tout appel à **divise** avec un deuxième argument égal à 0 va interrompre l'exécution de la fonction et provoquer l'affichage du message d'erreur suivant :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divise
ZeroDivisionError: integer division or modulo by zero
```

Pour définir plus proprement une fonction partielle, il est préférable de vérifier que les valeurs passées en argument sont bien dans son domaine d'utilisation. On peut pour cela vérifier une *précondition* à l'aide d'une instruction `assert e`. Cette instruction évalue l'expression booléenne *e* et provoque une erreur *AssertionError* si *e* est fausse. Ainsi, on peut ajouter une instruction `assert y != 0` à la fonction **divise** pour interrompre son exécution si *y* est égal à 0 :

```
def divise(x, y):
    assert y != 0
    return x // y
```

Le message d'erreur suivant est alors affiché dès que l'expression `y != 0` est évaluée à `False` :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divise
AssertionError
```

Afin d'afficher un message d'erreur plus informatif que *AssertionError*, on peut associer une deuxième expression à **assert**. Cette seconde expression sera évaluée uniquement si la première est fausse et sa valeur sera affichée pour compléter le message d'erreur. Ainsi, on peut écrire :

```
def divise(x, y):
    assert y != 0, 'division par 0 impossible'
    return x // y
```

Le message suivant sera affiché chaque fois que l'expression `y != 0` sera fausse :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divise
AssertionError: division par 0 impossible
```

Il est préférable de placer ces préconditions au *début* du corps de la fonction, afin de s'assurer qu'aucune instruction n'est exécutée si les arguments ne sont pas dans le domaine de

la fonction. Par exemple, on considère la fonction suivante, qui incrémente une variable globale `y` et ne s'assure que très tardivement de la valeur de son argument `x` :

```
def f(x):
    global y
    ...
    y = y + 1
    ...
    assert x != 0
    return 1 // x
```

Bien que chaque appel à `f(0)` provoque une erreur `AssertionError`, toutes les instructions placées avant `assert x != 0`, en particulier l'affection `y = y + 1`, seront exécutées. Ceci peut être une source d'erreur difficile à détecter dans un programme.

Une autre solution pour écrire une fonction partielle est de ne renvoyer aucun résultat quand les arguments sont en dehors du domaine. Par exemple, on peut écrire la fonction `divise` en utilisant une instruction conditionnelle qui teste la valeur de `y` avant de diviser :

```
def divise(x, y):
    if y != 0:
        return x // y
```

Comme on l'a vu dans la section 5.1.1, la valeur `None` sera renvoyée par `divise` si la variable `y` est égale à `0`. Cette solution présente tout de même l'inconvénient de ne pas informer clairement l'utilisateur quand la fonction est appelée en dehors de son domaine.

5.2.4 Fonctions de bibliothèque

Tous les langages de programmation répandus proposent des fonctions toutes faites pour la plupart des besoins courants, écrites par les concepteurs du langage eux-mêmes, ou par des utilisateurs, puis intégrées au fil du temps dans la distribution du langage. Ces fonctions écrites dans le langage et fournies avec lui sont regroupées dans ce qu'on appelle la *bibliothèque standard*.

Python n'échappe pas à la règle et prétend même être un langage « piles incluses » pour signifier que tout ce dont le programmeur peut avoir besoin est fourni avec le langage. Les différentes fonctions sont organisées en *modules* thématiques variés : représentation exacte des fractions, compression de fichiers, protocoles réseau….

Parmi les modules les plus utiles à l'élève de classes préparatoires — et, pour certains, déjà mentionnés plus haut — on peut citer :

- `math` qui contient toutes les fonctions habituellement utilisées en analyse ;
- `random` qui calcule des nombres pseudo-aléatoires ;
- `fractions` qui sert à manipuler des nombres rationnels en valeur exacte ;
- `numpy` qui fournit des outils variés pour le calcul scientifique. Ce dernier ne fait pas partie de la bibliothèque standard et doit donc être installé à part.

Pour utiliser les fonctions d'un module, on commence par importer ce dernier une fois pour toutes, en début de session interactive ou en début de programme. On prend ici l'exemple du module `math` :

```
In [7]: import math
```

Pour toute la suite de la session (ou du programme), on peut alors utiliser les fonctions de ce module en faisant précéder leur nom de celui du module :

```
In [8]: math.cos(14)

Out[8]: 0.1367372182078336
```

```
In [9]: math.sqrt(1 - math.sin(14)**2)

Out[9]: 0.13673721820783322
```

Cette notation permet au programmeur de définir par exemple sa propre fonction `sqrt`, sans risquer une homonymie avec celle fournie par le module. De même, plusieurs modules peuvent proposer des fonctions ayant le même nom, mais qui seront différenciées par leurs préfixes.

Une autre forme possible est de n'importer que la fonction dont on a besoin, par exemple `from math import sqrt`. On utilise alors directement `sqrt` sans le préfixe `math`, au prix d'un risque d'homonymie avec une fonction définie par ailleurs.

Enfin, les modules ne fournissent pas que des fonctions : certains proposent des constantes, de nouveaux types de données, etc.

```
In [10]: math.pi

Out[10]: 3.141592653589793
```

SAVOIR-FAIRE Rechercher une information au sein d'une documentation en ligne

Python propose près de deux cents modules, chacun susceptible de contenir jusqu'à plusieurs dizaines de fonctions. Il est donc exclu de les connaître tous, *a fortiori* de mémoriser leurs spécifications.

Un programmeur qui veut utiliser des fonctions de bibliothèque — et il y a intérêt s'il veut être efficace — doit donc savoir chercher l'information qui lui est nécessaire. Beaucoup de langages de programmation proposent une documentation officielle en ligne. Pour Python, il s'agit de `http://docs.python.org/`.

Les bibliothèques des langages sont organisées par thématiques, ce qui restreint assez vite l'espace de recherche. Ensuite, le caractère numérique d'une documentation en ligne fait qu'elle propose souvent des fonctions de recherche par mots-clés.

➡

Enfin, on veillera à consulter la documentation qui correspond à la version du langage que l'on utilise (pour Python, c'est particulièrement important à cause de l'incompatibilité entre les versions 2.x et 3.x).

5.2.5 Méthodes

Le langage Python appartient à la famille des langages dits *orientés objet*. Dans cet ouvrage, cet aspect du langage n'est pas du tout abordé. Néanmoins, l'accès à certaines bibliothèques oblige à introduire la notion de *méthode* liée à la programmation orientée objet.

Pour ce qui concerne cet ouvrage, une méthode n'est rien d'autre qu'une fonction *associée* à une valeur, telle qu'un entier, une chaîne de caractères, une liste, etc. Une valeur à laquelle sont associées des méthodes est appelée un *objet*. Par exemple, on prend `s` une chaîne de caractères définie par :

```
s = 'Ceci est une chaine'
```

On peut obtenir sa longueur de deux manières différentes. La première consiste à appeler la fonction `len`, avec la syntaxe :

```
In [11]: len(s)
Out[11]: 19
```

La seconde consiste à appeler sa méthode `__len__`, avec la syntaxe :

```
In [12]: s.__len__()
Out[12]: 19
```

Tout comme une fonction, une méthode peut prendre des arguments. Ainsi, on compte le nombre d'occurrences du caractère `'e'` dans la chaîne `s` avec la méthode `count` de `s` :

```
In [13]: s.count('e')
Out[13]: 4
```

D'une manière générale, la syntaxe d'appel d'une méthode est `objet.methode(arguments)`. Dans le contexte de cet ouvrage, ce n'est pas différent de l'appel d'une fonction où l'objet serait passé comme un argument supplémentaire, c'est-à-dire `methode(objet, arguments)`.

Voici deux autres exemples de méthodes sur les chaînes de caractères. L'opération permettant de passer d'une liste de caractères à la chaîne correspondante s'effectue par la méthode `join` d'une chaîne qui va servir de séparateur :

```
In [14]: ''.join(['a','b','c'])
Out[14]: 'abc'
```

```
In [15]: '-'.join(['d','e','f'])
Out[15]: 'd-e-f'
```

Cette méthode fonctionne également avec une liste de chaînes pour argument :

```
In [16]: ' '.join(['Ceci', 'est', 'une', 'phrase'])
Out[16]: 'Ceci est une phrase'
```

On réalise l'opération duale, c'est-à-dire découper une chaîne selon un séparateur donné, à l'aide la méthode `split` :

```
In [17]: 'Ceci est une phrase'.split('')
Out[17]: ['Ceci', 'est', 'une', 'phrase']
```

D'autres méthodes seront présentées dans la suite de cet ouvrage, notamment pour manipuler les piles et effectuer des entrées/sorties dans des fichiers.

5.3 La récursivité

Avertissement : les contenus abordés dans cette section sont au programme de seconde année uniquement. En particulier, certains exercices nécessitent d'avoir traité la complexité (section 6.1).

On considère la suite (u_n) suivante, qui calcule une approximation de $\sqrt{3}$:

$$\left\{ \begin{array}{lcl} u_0 & = & 2 \\ u_n & = & \dfrac{1}{2}\left(u_{n-1} + \dfrac{3}{u_{n-1}}\right) \end{array} \right.$$

On peut calculer cette suite à l'aide d'une fonction `u`, qui prend un argument *n*, et telle que `u(n)` renvoie la valeur de u_n. En Python, on peut écrire la fonction `u` en suivant *directement* la définition précédente :

```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

On remarque que dans la dernière ligne, il y a deux appels à la fonction pour calculer u_{n-1}. En effet, rien n'interdit d'appeler la fonction `u` à l'intérieur de son propre corps. On nomme alors cela une *fonction récursive*. Pour plus d'efficacité, on peut « factoriser » les deux appels `u(n-1)` en stockant le résultat dans une variable locale, ce qui supprime un appel potentiellement coûteux à `u` :

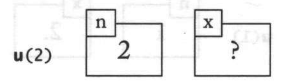
```
    ...
    else:
        x = u(n-1)
        return 0.5 * (x + 3. / x)
```

Ainsi, pour calculer une valeur approchée de $\sqrt{3}$, il suffit d'écrire le programme suivant :

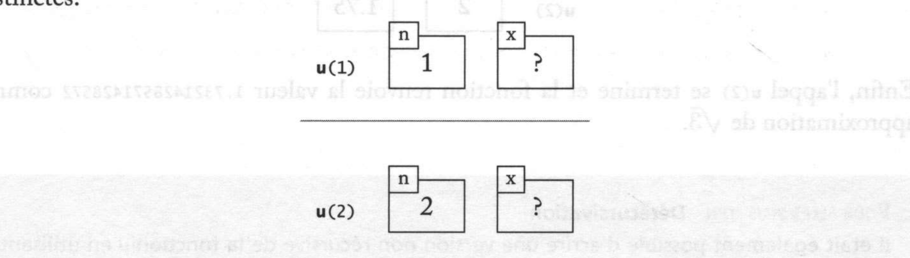
```
print(u(2))
```

Il affiche `1.7321428571428572` à l'écran. Pour comprendre comment ce résultat est calculé, on va suivre pas à pas l'exécution de cet appel de fonction.

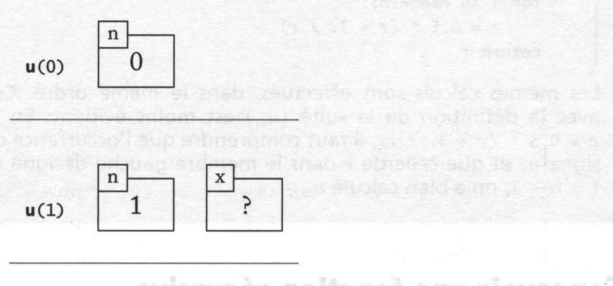
Juste après l'appel `u(2)`, la fonction `u` compare la valeur de son argument formel *n* (qui vaut ici `2`) avec `0` et exécute la branche **`else`** de l'instruction conditionnelle. Avant d'exécuter l'instruction `x = u(n-1)`, l'état de la mémoire est donc le suivant :



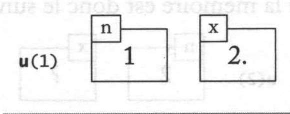
La variable locale *x* n'a toujours pas reçu de valeur. L'exécution se poursuit par l'appel `u(2-1)`. Toute la subtilité des fonctions récursives se dévoile dans ce deuxième appel. Que ce soit les paramètres formels ou les variables locales, toutes les variables dont la portée est limitée à une fonction s'ajoutent à l'état mémoire du programme lorsque celle-ci est appelée. Ainsi, l'état mémoire après ce deuxième appel est constitué de deux variables *n* et de deux variables locales *x* : celles allouées par l'appel `u(2)` et celles allouées pour `u(1)`. Ces variables n'ont de commun que le nom qu'on leur a donné, car elles représentent bien des cases mémoire distinctes.

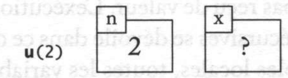


Tout comme pour l'appel précédent, la nouvelle variable locale *x* ne contient toujours pas de valeur avant d'exécuter `x = u(1-1)`. Ce dernier appel à la fonction `u` aboutit à l'état mémoire suivant :

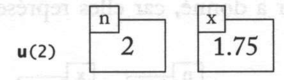


Seule une nouvelle case pour *n* est allouée en mémoire (puisque la branche `else` n'est pas exécutée). L'appel `u(0)` se termine alors par `return 2.`, ce qui a pour effet, non seulement de supprimer de la mémoire les variables locales allouées pour cet appel, mais également d'affecter la valeur `2.` à la variable *x* de l'appel précédent. On se trouve alors dans l'état mémoire suivant :





De la même manière, l'appel `u(1)` se termine par `return 0.5 * (2. + 3. / 2.)` et on revient à l'état mémoire du premier appel :



Enfin, l'appel `u(2)` se termine et la fonction renvoie la valeur `1.7321428571428572` comme approximation de $\sqrt{3}$.

POUR ALLER PLUS LOIN Dérécursivation

Il était également possible d'écrire une version non récursive de la fonction `u` en utilisant une boucle, par exemple sous la forme suivante :

```
def u(n):
    r = 2.
    for i in range(n):
        r = 0.5 * (r + 3. / r)
    return r
```

Les mêmes calculs sont effectués, dans le même ordre. Cependant, le rapprochement avec la définition de la suite (u_n) est moins évident. En particulier, dans l'instruction `r = 0.5 * (r + 3. / r)`, il faut comprendre que l'occurrence de *r* dans le membre droit désigne u_i et que celle de *r* dans le membre gauche désigne u_{i+1}. Puisqu'on termine avec $i = n - 1$, on a bien calculé u_n.

5.3.1 Concevoir une fonction récursive

La conception d'une fonction récursive n'est pas éloignée du principe de *démonstration par récurrence*. Par exemple, le principe de récurrence simple permet de démontrer une propriété P_n pour tout $n \in \mathbb{N}$ en démontrant d'une part le cas de base P_0 et d'autre part que P_{n-1} implique P_n pour tout $n > 0$.

De la même façon, on peut définir une fonction `f` prenant en argument un entier naturel *n* en se ramenant au calcul de `f(0)` d'une part et de `f(n)` en fonction de `f(n-1)` d'autre part. La fonction `f` prend alors la forme suivante :

```
def f(n):
    if n == 0:
        return ...
    else:
        return ... f(n-1) ...
```

L'exemple le plus classique est sûrement celui de la fonction `factorielle`, dont la définition est donnée page 104 :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n-1)
```

Il est important de noter qu'une telle fonction ne terminera pas sur un argument *n* négatif. En effet, `factorielle(-1)` appellerait `factorielle(-2)`, qui appellerait `factorielle(-3)`, etc. Il s'agit donc d'une fonction partielle, à laquelle on peut appliquer toute solution discutée dans la section 5.2.3. En particulier, on peut s'assurer que *n* est bien un entier naturel en écrivant :

```
def factorielle(n):
    assert n >= 0
    ...
```

Le schéma de récurrence simple peut être appliqué à des fonctions ayant d'autres arguments que *n*. Ainsi, la fonction `puissance` qui calcule *x* à la puissance *n* peut facilement être définie par récurrence simple sur *n*, de la manière suivante :

```
def puissance(x, n):
    if n == 0:
        return 1
    else:
        return x * puissance(x, n-1)
```

Comme en mathématiques, le schéma de récurrence simple peut être adapté à des définitions impliquant plusieurs cas de base. Ainsi, on évite une multiplication inutile quand *n* vaut `1` dans la fonction `puissance` en la réécrivant de la façon suivante :

```
def puissance(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * puissance(x, n-1)
```

L'écriture de fonctions récursives n'est pas limitée au schéma de récurrence simple. On peut également utiliser un schéma de récurrence forte, c'est-à-dire effectuer des appels récursifs sur des valeurs strictement inférieures à `n-1`. Reprenant l'exemple du calcul de x^n, on propose un meilleur algorithme qui exploite les deux identités suivantes :

$$\begin{cases} x^{2k} &= (x^k)^2 \\ x^{2k+1} &= x(x^k)^2 \end{cases}$$

Elles permettent de ramener le calcul de x^n à celui de $x^{\lfloor \frac{n}{2} \rfloor}$. Le cas de base reste le même, à savoir $x^0 = 1$. Dans le cas récursif, on commence par calculer $x^{\lfloor \frac{n}{2} \rfloor}$ dans une variable *r*, puis on teste la parité de *n* pour choisir entre les deux identités ci-avant. Finalement, on obtient le code suivant :

```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

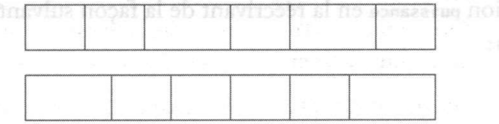
Ainsi, le calcul de `puissance_rapide(3, 5)` se ramène directement au calcul de `puissance_rapide(3, 2)` et évite les appels à `puissance_rapide(3, 3)` et `puissance_rapide(3, 4)`.

Exercice 5.9 * Écrire une variante (toujours récursive) de la fonction **puissance_rapide** qui exploite plutôt les identités suivantes :

$$\begin{cases} x^{2k} &= (x^2)^k \\ x^{2k+1} &= x(x^2)^k \end{cases}$$

Y a-t-il une différence dans le nombre de multiplications effectuées ?

Pour continuer l'analogie avec le principe de récurrence forte en mathématiques, il est parfois nécessaire d'effectuer *plusieurs* appels récursifs pour calculer `f(n)`. On considère le problème consistant à calculer le nombre de façons de construire une rangée de longueur n avec des briques de longueur 2 et 3. Voici par exemple deux rangées de longueur $n = 14$.



On peut en dénombrer 21 au total. Les cas de base correspondent à $n = 1$ (pas de solution) et $2 \leqslant n \leqslant 3$ (une solution unique). Le calcul récursif consiste à se ramener au cas $n - 2$ (ajout d'une brique de longueur 2) et au cas $n - 3$ (ajout d'une brique de longueur 3).

```
def briques(n):
    assert n >= 1
    if n == 1:
        return 0
    elif n == 2 or n == 3:
        return 1
    else:
        return briques(n-2) + briques(n-3)
```

L'exercice 5.24 propose un autre exemple.

Enfin, il est parfois possible que la définition d'une fonction `f` fasse appel à une fonction `g`, et que la définition de `g` fasse appel elle-même à celle de `f`. On parle alors de fonctions *mutuellement récursives*. Par exemple, on peut définir une fonction `pair` pour déterminer si un entier `n` est pair par récurrence mutuelle avec une fonction `impair` qui, elle, détermine si un entier `n` est impair. En Python, il suffit d'écrire ces deux fonctions, l'une à la suite de l'autre :

```
def pair(n):
    return (n == 0) or impair(n-1)

def impair(n):
    return (n != 0) and pair(n-1)
```

5.3.2 Terminaison et correction d'une fonction récursive

Dans cette section, on va montrer comment raisonner à propos d'une fonction récursive, pour démontrer d'une part sa terminaison et d'autre part sa correction, c'est-à-dire le fait qu'elle calcule bien ce qu'elle doit calculer. Sans surprise, on utilisera le principe de démonstration par récurrence pour démontrer la correction d'une fonction récursive.

On va prendre l'exemple de la fonction `factorielle` définie plus haut page 133. On veut montrer par récurrence sur $n \geqslant 0$ la propriété suivante :

$$H_n : \texttt{factorielle}(n) \text{ termine et renvoie la valeur } n!$$

La propriété H_0 est vérifiée car `factorielle(0)` se réduit à `return 1`. Pour $n > 0$, on suppose H_{n-1} et on cherche à montrer H_n. Le calcul de `factorielle(n)` commence par un appel récursif à `factorielle(n-1)`. Par hypothèse de récurrence, cet appel termine et renvoie la valeur $(n-1)!$. Puis l'appel à `factorielle(n)` multiplie ce résultat par `n` et renvoie le produit. Donc, cet appel termine et renvoie bien $n \times (n-1)! = n!$, ce qui démontre H_n.

Il est important de noter que nous n'avons rien démontré quant aux appels à la fonction `factorielle` sur des arguments négatifs. En particulier, ils peuvent ne pas terminer (ce qui est le cas ici), renvoyer des valeurs farfelues, ou encore échouer (ce qui serait le cas avec `assert n >= 0` par exemple).

Si on a utilisé un schéma de récurrence forte pour définir une fonction récursive, alors il faudra bien entendu en démontrer la correction par récurrence forte également. Avec l'exemple de la fonction `puissance_rapide` page 134, on cherche à montrer par récurrence forte sur $n \geqslant 0$ la propriété suivante :

$$H_n : \texttt{puissance_rapide}(x, n) \text{ termine et renvoie la valeur } x^n$$

La propriété H_0 est vérifiée car `puissance_rapide(x, 0)` se réduit à `return 1` (en admettant que l'on a posé $0^0 = 1$ arbitrairement). Soit maintenant $n > 0$; on suppose H_i pour tout $0 \leqslant i < n$ et on veut montrer H_n. Le calcul de `puissance_rapide(x, n)` commence par un appel récursif `r = puissance_rapide(x, n // 2)`. On pose $k = \lfloor \frac{n}{2} \rfloor$. Comme $n > 0$, on a $k < n$. On peut donc appliquer l'hypothèse de récurrence H_k, qui affirme que l'appel `puissance_rapide(x, n // 2)` termine et renvoie la valeur x^k. On distingue alors deux cas, selon la parité de n. Si n est pair, c'est-à-dire $n = 2k$, alors le programme effectue `return r * r`. Donc il termine et renvoie $x^k \times x^k = x^{2k} = x^n$, ce qui démontre H_n. On procède de même lorsque n est impair.

Exercice 5.10 ** Démontrer la terminaison et la correction de la fonction `puissance` page 133.

Exercice 5.11 Montrer la correction de la fonction `briques` définie plus haut (page 134).

Exercice 5.12 ** Que se passe-t-il avec la fonction `puissance_rapide` si on écrit `n / 2` au lieu de `n // 2` (en Python 3) ou encore `n / 2.` ?

ATTENTION Limitation de la récursivité en Python

Le langage Python limite, arbitrairement, le nombre d'appels imbriqués à 1 000. Une fonction qui fait plus de 1 000 appels récursifs provoque l'erreur suivante :

```
RuntimeError: maximum recursion depth exceeded
```

Même si cette limite semble basse, elle n'exclut pas pour autant l'utilisation de fonctions récursives en Python. En effet, il existe de nombreuses situations où l'on sait que le nombre d'appels sera bien inférieur à 1 000. C'est le cas en particulier pour des fonctions qui font un nombre d'appels *logarithmique* en la taille des données. Voir par exemple l'exercice 5.15.

POUR ALLER PLUS LOIN Démontrer la terminaison d'une fonction récursive

De même qu'on peut justifier la terminaison d'une boucle en exhibant un entier naturel qui décroît strictement à chaque itération (voir page 98), on peut démontrer la terminaison d'une boucle récursive en exhibant en entier naturel qui décroît strictement à chaque appel récursif. Le plus souvent, il s'agira directement de l'un des arguments de la fonction récursive, comme dans le cas des fonctions définies plus haut dans cette section.

Cependant, rien n'exclut qu'il s'agisse d'une expression plus compliquée. La célèbre « fonction 91 » de McCarthy en est un exemple :

```
def f91(n):
    if n <= 100:
        return f91(f91(n + 11))
    else:
        return n - 10
```

Il existe même des fonctions récursives dont on ne sait pas démontrer la terminaison, comme la célèbre suite de Syracuse :

```
def syracuse(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return syracuse(n // 2)
    else:
        return syracuse(3 * n + 1)
```

Exercice 5.13 Démontrer la terminaison de la fonction `f91`.

SAVOIR-FAIRE Déboguer une fonction récursive

Lorsqu'on débogue une fonction récursive, il faut évidemment entrer dans les appels récursifs pour comprendre ce qui se passe : on est naturellement conduit à effectuer *Pas vers l'intérieur*.

Cependant, comme expliqué dans ce chapitre, à chaque appel récursif, il est créé un nouvel ensemble de variables locales à la fonction. L'explorateur de variables ne montre que les variables en cours d'utilisation, autrement dit celles qui sont propres à l'appel dans lequel on se situe.

Il y a cependant une possibilité pour visualiser la suite d'appels récursifs déjà effectués. En tapant `w` (pour `where`) dans l'interpréteur, on voit apparaître la liste des appels de fonctions en cours, le plus récent étant situé tout en bas. Ensuite, il est possible de se déplacer dans cette liste avec les commandes `u` (pour **up**) et `d` (pour **down**). L'explorateur de variables montre alors les contenus des variables correspondant à ces différents appels.

Spyder ne propose pour l'instant pas d'interface graphique pour effectuer ces manipulations.

5.3.3 Complexité d'une fonction récursive

On explique ici comment calculer le *coût* d'une fonction récursive, à savoir le nombre d'opérations élémentaires qu'elle effectue ou son occupation mémoire totale. La notion de complexité sera présentée plus en détail dans le prochain chapitre, section 6.1.

Reprenons l'exemple de la fonction `u` (définie p. 130) :

```
def u(n):
    if n == 0:
        return 2.
    else:
        x = u(n-1)
        return 0.5 * (x + 3. / x)
```

et évaluons le nombre d'opérations arithmétiques (addition, multiplication et division) qu'elle effectue.

Si n désigne la valeur de son argument, notons $C(n)$ ce nombre d'opérations. En suivant la définition de la fonction `u`, on obtient les deux équations suivantes :

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n-1)+3 \end{aligned}$$

En effet, dans le cas $n = 0$, on ne fait aucune opération arithmétique. Et dans le cas $n > 0$, on fait d'une part un appel récursif sur la valeur $n-1$, d'où $C(n-1)$ opérations, puis trois opérations arithmétiques (une multiplication, une addition et une division). Il s'agit d'une suite arithmétique de raison 3, dont le terme général est :

$$C(n) = 3n$$

Le nombre d'opérations arithmétiques effectuées par la fonction `u` est donc proportionnel à n.

Si en revanche on avait écrit la fonction `u` plus naïvement, avec deux appels récursifs `u(n-1)`, c'est-à-dire :

```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

alors les équations définissant $C(n)$ seraient les suivantes :

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n-1)+C(n-1)+3 \end{aligned}$$

En effet, il convient de prendre en compte le coût $C(n-1)$ des deux appels à `u(n-1)`. Il s'agit maintenant d'une suite arithmético-géométrique, dont le terme général est :

$$C(n) = 3(2^n - 1)$$

Une autre manière d'évaluer le coût d'une fonction récursive est de calculer le nombre d'*appels*, puis d'évaluer le coût de chaque appel. Si on note $A(n)$ le nombre d'appels récursifs dans les deux exemples précédents, on a :

$$\begin{array}{rcl} A(0) & = & 0 \\ A(n) & = & A(n-1)+1 \end{array}$$

dans le premier cas, et :

$$\begin{array}{rcl} A(0) & = & 0 \\ A(n) & = & A(n-1)+A(n-1)+1 \end{array}$$

dans le second cas. Le terme général est donc $A(n) = n$ dans le premier cas et $A(n) = 2^n - 1$ dans le second. Puisqu'on n'a ici aucune opération arithmétique dans le cas de base $n = 0$ et exactement trois opérations arithmétiques dans le cas récursif, on retrouve immédiatement la valeur de $C(n)$ calculée précédemment. D'une manière générale, la valeur de $C(n)$ ne se déduit pas toujours aussi facilement de la valeur de $A(n)$. En effet, il peut y avoir des opérations dans le cas de base et/ou un nombre d'opérations arithmétiques variant selon la valeur de n dans le cas récursif.

Comme nous l'avons expliqué page 131, chaque appel récursif alloue de la mémoire pour les paramètres effectifs et les variables locales de cet appel. L'occupation mémoire d'un calcul récursif admet donc pour majorant le produit du nombre d'appels récursifs par la quantité de mémoire allouée par chaque appel. Dans les deux exemples précédents, on a calculé explicitement le nombre d'appels $A(n)$. L'occupation mémoire est donc $2n$ dans le premier cas (il y a deux cases mémoire, une pour *n* et une autre pour *x*) et $2^n - 1$ dans le second cas (il y a une case mémoire, pour *n*). Cependant, dans le second cas, les $2^n - 1$ cases mémoire ne seront pas utilisées simultanément. En effet, celles allouées pour le premier appel à `u(n-1)` peuvent être réutilisées pour le second (et en pratique elles le sont). Pour une analyse plus fine de l'occupation mémoire, il convient donc de calculer le nombre d'appels *imbriqués*.

Exercice 5.14 On considère la première version de la fonction **`puissance`**, définie p. 133.
1 Combien effectue-t-elle exactement d'appels récursifs pour calculer x^n ?
2 Quel est son coût en mémoire ?

Exercice 5.15 * On considère la fonction **`puissance_rapide`** définie p. 134.
1 Montrer qu'elle calcule x^n en effectuant un nombre total d'appels récursifs proportionnel à $\log n$.
2 A-t-on la même complexité quand on n'utilise pas de variable locale *r*, mais que l'on écrit directement : `puissance_rapide(x, n // 2) * puissance_rapide(x, n // 2)` à la place de `r * r` ?

5.4 Exercices

Exercice 5.16 Écrire en Python une fonction qui prend comme argument un entier n et renvoie l'entier 2^n.

Exercice 5.17 * Écrire en Python une fonction qui prend comme argument un entier n et renvoie un booléen qui indique si cet entier est premier ou non.

Exercice 5.18 Qu'affiche le programme suivant ? Pourquoi ?

```
def g(x):
    global a
    a = 10
    return 2 * x

def f(x):
    v = 1
    return g(x) + v

a = 3
print(f(6)+a)
```

Proposer une adaptation de ce programme dans laquelle le résultat affiché ne dépend pas de l'ordre d'évaluation.

Exercice 5.19 Qu'affiche le programme suivant ? Pourquoi ?

```
def g(x):
    global v
    v = 1000
    return 2 * x

def f(x):
    v = 1
    return g(x) + v

a = 3
print(f(6)+a)
```

Proposer une adaptation de ce programme dans laquelle le résultat affiché ne dépend pas de l'ordre d'évaluation.

Exercice 5.20 Soit `f` la fonction suivante :

```
def f(x):
    while (True):
        x = x + 1
        if (x == 1000):
            return 2 * x
```

Que renvoie l'appel `f(500)` ?

Exercice 5.21 ** Écrire une fonction qui calcule le PGCD des deux entiers naturels passés en arguments en suivant l'algorithme d'Euclide, à l'aide d'une boucle `while`. Démontrer la correction et la terminaison de cette fonction.

Exercice 5.22 On écrit un programme qui gère automatiquement le score au tennis.

1 Écrire une condition booléenne qui permet de décider si un joueur a gagné un jeu.

2 Définir une fonction qui compte les points au cours d'un jeu. En entrée, on demande répétitivement quel joueur, 1 ou 2, gagne le point ; au fur et à mesure, on calcule et on affiche le score. Le programme s'arrête dès qu'un joueur gagne le jeu, après avoir affiché le nom du vainqueur.

3 Pour faciliter les tests, écrire une seconde version de cette fonction avec pour seule entrée une chaîne de caractères qui contient les numéros successifs des joueurs marquant les points ; la fonction lit cette chaîne caractère par caractère pour compter les points. Ainsi, l'entrée `211222` sera comprise comme : le joueur 2 gagne un point, puis le joueur 1 en gagne deux, puis le joueur 2 en gagne trois et le joueur 2 gagne donc le jeu.

4 Écrire une deuxième fonction qui compte les jeux au cours d'un set et s'arrête lorsqu'un joueur gagne le set. Cette fonction fera appel à la précédente pour savoir qui gagne les jeux. On n'oubliera pas de prévoir le cas particulier du jeu décisif.
5 Écrire une troisième fonction qui compte les sets et s'arrête lorsqu'un joueur gagne le match. On pourra, avant de commencer le match, demander en combien de sets gagnants il est joué.

Exercices utilisant la récursivité

Exercice 5.23 Écrire une fonction récursive qui calcule la somme des n premiers entiers. Quelle est la complexité de cette fonction ?

Exercice 5.24 * Suite de Fibonacci. On considère la suite de Fibonacci définie par :

$$\begin{cases} F_0 & = & 0 \\ F_1 & = & 1 \\ F_n & = & F_{n-2} + F_{n-1} \text{ pour } n \geqslant 2. \end{cases}$$

Écrire une fonction récursive basée sur ces relations qui prend n en argument et renvoie F_n. Quelle est sa complexité ?

Accélérer le calcul de F_n en écrivant plutôt une fonction récursive auxiliaire qui prend en arguments F_{n-1}, F_n et $k \geqslant 0$ et renvoie F_{n+k} (on pourra poser $F_{-1} = 1$). Quelle est la nouvelle complexité ?

Exercice 5.25 ** Écrire une fonction récursive qui calcule le PGCD des deux entiers naturels passés en arguments en suivant l'algorithme d'Euclide. Démontrer la terminaison et la correction de cette fonction.

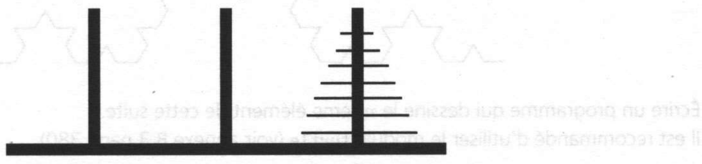
Exercice 5.26 Modifier la fonction qui calcule les termes de la suite de Syracuse (voir l'encadré page 137) pour qu'elle affiche les éléments de la suite jusqu'au premier rang n_0 tel que $u_{n_0} = 1$, ainsi que la valeur de n_0.

Exercice 5.27 Définir une fonction récursive qui, étant donné un entier n, décide si l'écriture en base 3 de n ne comporte que des 0 et des 1. Quelle est la complexité de cette fonction ?

Exercice 5.28 ** Les tours de Hanoï. *Les tours de Hanoï* est un jeu inventé par Édouard Lucas en 1883. Il est formé de sept disques de tailles différentes répartis en trois colonnes. Au départ, tous les disques sont empilés sur la colonne de gauche par taille croissante.



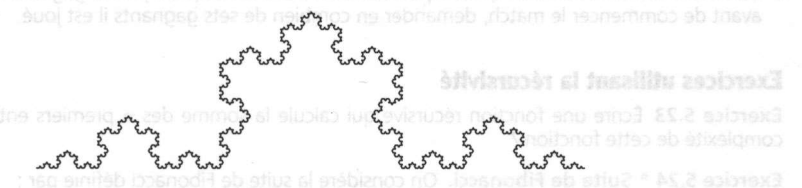
Les seuls mouvements possibles sont les déplacements d'un disque situé au sommet d'une colonne vers le sommet d'une autre colonne, à condition que la colonne d'arrivée soit vide ou que le disque déplacé soit plus petit que son sommet. On note `n -> n'` le déplacement d'un disque de la colonne `n` vers la colonne `n'`. Le but du jeu est de déplacer tous les disques vers la colonne de droite.



Écrire un programme qui affiche une solution du jeu sous la forme d'une suite de mouvements.

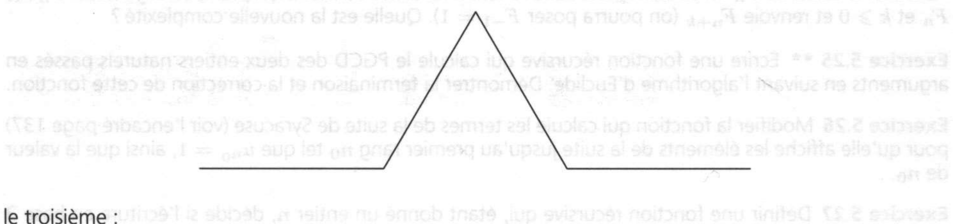
Indication : dans une variante, le jeu n'est formé que de six disques. Si l'on sait résoudre le jeu à six disques, comment résoudre le jeu à sept disques ?

Exercice 5.29 * Le flocon de Von Koch. *Le flocon*, courbe définie par Helge Von Koch en 1906, est un exemple de courbe continue partout et dérivable nulle part. C'est aussi un exemple d'ensemble fractal, c'est-à-dire dont la dimension de Hausdorff n'est pas un nombre entier.

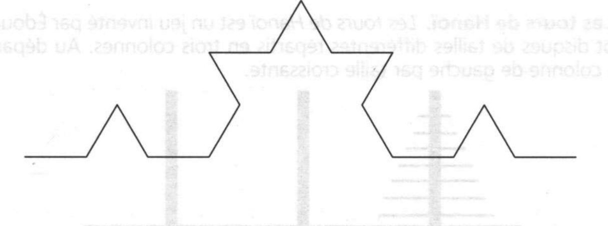


Cette courbe se définit comme la limite de la suite de courbes dont le premier élément est un segment :

et dans laquelle on passe d'un élément au suivant en divisant chaque segment en trois et en remplaçant le morceau du milieu par deux segments formant un triangle équilatéral avec le segment supprimé. Le deuxième élément est donc :



le troisième :



et le quatrième :

Écrire un programme qui dessine le n-ième élément de cette suite.
Il est recommandé d'utiliser le module **`turtle`** (voir annexe B.3 page 380).

6

Notions de complexité et algorithmique sur les tableaux

Dans ce chapitre, nous présentons la notion de complexité, qui décrit le temps et la mémoire nécessaires pour exécuter un algorithme et caractérise donc son efficacité.

Nous présentons ensuite la structure de tableau, ainsi que certaines opérations courantes qui s'y appliquent. Nous étudions la complexité de tous les algorithmes présentés.

6.1 Complexité d'un algorithme

6.1.1 Plusieurs algorithmes pour un même problème

Pour traiter un même problème, il existe souvent plusieurs algorithmes. Quand on doit choisir, l'un des critères est celui du *temps d'exécution*, qu'on appelle aussi parfois *coût* de l'algorithme. Cette dénomination n'est pas usurpée car ce temps conditionne les ressources utilisées (machine sur laquelle on exécutera l'algorithme, consommation électrique...).

Pour un logiciel interactif, par exemple, un temps de réponse court est un élément essentiel du confort de l'utilisateur. De même, certains programmes industriels doivent être utilisés un grand nombre de fois dans un délai très court et même un programme qui n'est exécuté qu'une seule fois, par exemple un programme de simulation écrit pour tester une hypothèse de recherche, est inutilisable s'il demande des mois ou des années de calcul.

Par exemple, si l'on cherche à afficher la liste des diviseurs d'un nombre entier n, on peut écrire l'algorithme naïf suivant :

```
def diviseurs(n):
    for i in range(1,n+1):
        if n % i == 0:
            print(i)
```

Cet algorithme réalise exactement n calculs de restes de divisions euclidiennes, n comparaisons et au plus n affichages. Cependant, on peut aussi utiliser le fait que si $n = p \times q$ avec $p \geqslant \sqrt{n}$, alors q est un diviseur de n inférieur ou égal à $\sqrt{n}$. Il suffit donc de chercher chaque diviseur q inférieur ou égal à $\sqrt{n}$ et de calculer $p = n/q$ pour obtenir tous les diviseurs. Dans le cas où n est un carré parfait, on prend soin de ne pas afficher sa racine carrée deux fois :

```
import math
def diviseurs(n):
    for i in range(1,int(math.sqrt(n))+1):
        if n % i == 0:
            print(i)
            if n//i != i:
                print(n//i)
```

Ce deuxième algorithme ne fait plus que $\lfloor\sqrt{n}\rfloor$ itérations, qui effectuent chacune un calcul de reste, une ou deux comparaisons, et zéro, un ou deux affichage(s). Au total, il coûte donc un calcul de racine carrée, $\lfloor\sqrt{n}\rfloor$ calculs de reste, entre $\lfloor\sqrt{n}\rfloor$ et $2\lfloor\sqrt{n}\rfloor$ comparaisons et entre 0 et $2\lfloor\sqrt{n}\rfloor$ affichages.

En général, on cherche à déterminer comment le temps d'exécution d'un algorithme varie en fonction d'un paramètre qu'on appelle la *taille* du problème. Le temps de recherche des diviseurs d'un entier n dépend de n, qu'on pourra donc naturellement prendre comme taille du problème. Comme on l'a vu, selon l'algorithme, ce temps peut être proportionnel à n ou à $\sqrt{n}$. De même, quand on s'interrogera sur l'efficacité d'un algorithme manipulant des tableaux, on cherchera à comprendre comment le temps d'exécution de cet algorithme varie

en fonction du nombre d'éléments de ce tableau. Une autre possibilité pour la taille du problème est de considérer la taille de la représentation des données passées à ce programme. Par exemple, pour un programme traitant un texte, on prend pour taille du problème le nombre de caractères de ce texte.

ATTENTION Taille d'un entier

Lorsqu'un problème dépend d'un paramètre considéré n, il y a deux choix naturels pour la taille du problème : l'entier lui-même ou la taille des données nécessaires à sa représentation, c'est-à-dire son nombre de chiffres. Dans le problème précédent, si n est l'entier dont on cherche les diviseurs, il est nécessaire d'effectuer n ou $\sqrt{n}$ calculs de restes. S'il s'agit en revanche de la taille de l'entier dont on cherche les diviseurs, cela signifie que l'entier considéré possède n chiffres, donc qu'il est compris entre 10^{n-1} et 10^n. Il est donc nécessaire d'effectuer entre 10^{n-1} et 10^n calculs de restes dans le premier cas et d'en effectuer entre $10^{\frac{n-1}{2}}$ et $10^{\frac{n}{2}}$ dans le second.

L'évaluation du temps mis par un algorithme pour s'exécuter est un domaine de recherche à part entière, car elle se révèle parfois très difficile. Néanmoins, dans de nombreux cas, cette évaluation peut se faire en appliquant quelques règles simples.

SAVOIR-FAIRE Déterminer le coût d'un algorithme

Pour déterminer le coût d'un algorithme, on se fonde en général sur le *modèle de complexité* suivant :

- Une affectation, une comparaison ou l'évaluation d'une opération arithmétique ayant en général un faible temps d'exécution, celui-ci sera considéré comme l'unité de mesure du coût d'un algorithme.
- Le coût des instructions `p` et `q` en séquence est la somme des coûts de l'instruction `p` et de l'instruction `q`.
- Le coût d'un test `if b: p else: q` est inférieur ou égal au maximum des coûts des instructions `p` et `q`, plus le temps d'évaluation de l'expression `b`.
- Le coût d'une boucle `for i in iterable : p` est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction `p` si ce dernier ne dépend pas de la valeur de `i`. Quand le coût du corps de la boucle dépend de la valeur de `i`, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de `i`.
- Le cas des boucles `while` est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu *a priori*. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison (voir Savoir-faire *Démontrer qu'une boucle se termine* p. 101) et ainsi majorer le coût de l'exécution de la boucle.

Le cas particulier des boucles imbriquées illustre bien le principe de calcul du coût de la boucle `for`. Ainsi, si les deux boucles sont répétées respectivement m et m' fois, alors le corps de la boucle interne est exécuté $m \times m'$ fois en tout. Il est en effet répété à cause de cette boucle interne, mais aussi parce qu'elle-même est répétée dans son intégralité.

Exercice 6.1 avec corrigé Reprendre les programmes de l'exercice 4.31 et évaluer leur coût.

L'opération de base ici consiste à afficher une étoile à l'écran (ce qu'on peut assimiler à une affectation qui modifie l'état de la carte graphique au lieu de l'état de la mémoire).

Dans le premier programme, les nombres d'itérations des boucles interne et externe sont indépendants et ils sont respectivement de p *et* n. *Au total, on effectue* $n \times p$ *affichages dans ce programme, ce qu'on peut voir dans le résultat produit à l'écran.*

Dans le second programme, la boucle interne dépend du compteur de la boucle externe. Le nombre d'affichages effectués est donc : $\sum_{i=0}^{n-1}(i+1) = \frac{n(n+1)}{2}$

Pour être complet, il faudrait comptabiliser également l'affichage des retours chariot ; ils sont cependant nettement moins nombreux que les étoiles et ne jouent pas un rôle significatif dans le temps d'exécution du programme.

POUR ALLER PLUS LOIN Les limites de ce modèle de complexité

Le modèle de complexité qu'on a donné, comme tout modèle, n'est qu'un reflet imparfait de la réalité. Il n'est évidemment utile que dans les cas où il est suffisamment proche de la réalité.

Malheureusement, dans certains cas, les hypothèses sous-jacentes à ce modèle ne tiennent pas. Ainsi, les entiers n'étant pas bornés, il est irréaliste de penser qu'une opération arithmétique ait un coût unitaire : par exemple, l'addition de deux nombres entiers à n chiffres nécessite de lire tous leurs chiffres et d'écrire ceux du résultat et demande donc un temps de calcul proportionnel à n. Le temps de calcul d'une opération sur des entiers longs n'est pas une bonne unité de mesure, puisqu'il peut lui-même dépendre de la taille des opérandes.

Dans cet ouvrage, sauf mention expresse du contraire, on restera sur le modèle précédemment proposé, d'une part parce qu'il convient bien à une large classe de problèmes et d'autre part parce que le compliquer dépasserait le cadre du programme de cet enseignement.

6.1.2 Complexité et notation O

Dans l'exemple précédent, on a évalué de façon relativement fine le nombre des opérations effectuées par chacun des algorithmes, par exemple « n comparaisons » ou « entre $\sqrt{n}$ et $2\sqrt{n}$ comparaisons ». En réalité, lorsqu'on cherche à évaluer l'efficacité d'un algorithme, il est souvent inutile d'aller jusqu'à ce niveau de détail : on se contentera de dire que le nombre d'opérations élémentaires effectuées est par exemple proportionnel à n ou à $\sqrt{n}$.

Il y a plusieurs raisons à cela. D'une part, les différentes opérations élémentaires considérées ne demandent pas toutes exactement le même temps de calcul et cachent donc un facteur multiplicatif, borné mais très compliqué à déterminer précisément. D'autre part, le même programme peut être exécuté sur deux machines différentes, l'une étant par exemple

deux fois plus rapide que l'autre. Cela ne change évidemment rien à l'efficacité intrinsèque de l'algorithme et ce qui nous intéresse réellement n'est pas le temps précis d'exécution d'un programme, mais l'ordre de grandeur de ce temps en fonction de la taille des données.

Une dernière notion à considérer est celle du terme dominant dans le temps d'exécution d'un algorithme. Par exemple, si on a déterminé que ce temps était proportionnel à n^2+3n, dès que la taille n des données devient un peu importante, il est connu que le terme $3n$ augmente beaucoup moins vite que n^2 : on dit qu'il est négligeable devant ce dernier. Pour décrire l'efficacité d'un algorithme, seul le terme qui croît le plus vite a donc un intérêt. Par exemple, ici pour $n \geqslant 3$, on a $n^2+3n \leqslant 2n^2$; la quantité n^2+3n est donc bornée, à partir d'un certain rang, par le produit de n^2 et d'une constante. On dit alors que la quantité de n^2+3n est « un grand O de n^2 » et on écrira $n^2+3n = O(n^2)$. De manière générale, on dira qu'un algorithme a une *complexité* en $O(f(n))$ si son coût est, à partir d'un certain rang, inférieur au produit de $f(n)$ par une constante [1].

On va ébaucher un rapide inventaire des complexités qu'on pourra être amené à rencontrer. Il n'est pas possible, sur le plan théorique, de dire combien de temps un algorithme en $O(n)$ met à s'exécuter pour une valeur particulière de n, puisque deux algorithmes dont les temps de calcul seraient respectivement $n \times 10^{-9}s$ et $n \times 10^{9}s$ seraient tous les deux en $O(n)$, bien que le rapport de leurs temps d'exécution soit 10^{18}. Cependant, on peut donner les ordres de grandeur des temps d'exécution que l'on rencontre en pratique pour un problème de taille $n = 10^6$ sur un ordinateur personnel effectuant un milliard d'opérations par seconde.

Tableau 6.1 Ordres de grandeur des temps d'exécution d'un problème de taille 10^6 sur un ordinateur à un milliard d'opérations par seconde

	Nom courant	Temps pour $n = 10^6$	Remarques
$O(1)$	temps constant	1 ns	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !
$O(\log n)$	logarithmique	10 ns	En pratique, cela correspond à une exécution quasi instantanée. Bien souvent, à cause du codage binaire de l'information, c'est en fait la fonction $\log_2 n$ qu'on voit apparaître ; mais comme la complexité est définie à un facteur près, la base du logarithme n'a pas d'importance.
$O(n)$	linéaire	1 ms	Le temps d'exécution d'un tel algorithme ne devient supérieur à une minute que pour des données de taille comparable à celle des mémoires vives disponibles actuellement. Le problème de la gestion de la mémoire se posera donc avant celui de l'efficacité en temps.
$O(n^2)$	quadratique	1/4 h	Cette complexité reste acceptable pour des données de taille moyenne ($n < 10^6$), mais pas au-delà.

1. Le sens de cette notation sera précisé en cours de mathématiques.

Tableau 6.1 Ordres de grandeur des temps d'exécution d'un problème de taille 10^6 sur un ordinateur à un milliard d'opérations par seconde (suite)

$O(n^k)$	polynômiale	30 ans si $k = 3$	Ici, n^k est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(2^n)$	exponentielle	plus de $10^{300\,000}$ milliards d'années	Un algorithme d'une telle complexité est impraticable sauf pour de très petites données ($n < 50$). Comme pour la complexité logarithmique, la base de l'exponentielle ne change fondamentalement rien à l'inefficacité de l'algorithme.

Exercice 6.2 avec corrigé Les algorithmes suivants calculent et affichent différentes listes de nombres. Quelle est la complexité de chacun d'entre eux ?

```
def table1(n):
    for i in range(11):
        print(i * n)
```

```
def table2(n):
    for i in range(n):
        print(i * i)
```

```
def table3(n):
    for i in range(n):
        for j in range(n):
            print(i * j, end=" ")
        print()
```

- *L'algorithme* **table1** *affiche la table de multiplication de n jusqu'au rang 10. La boucle est toujours exécutée 11 fois et ne comporte qu'une multiplication. Le temps d'exécution ne dépend donc pas de l'entrée n : la complexité est $O(1)$.*
- *L'algorithme* **table2** *affiche la suite des carrés des nombres entiers jusqu'à $(n-1)^2$. La boucle est exécutée n fois et ne comporte qu'une multiplication : la complexité est $O(n)$.*
- *L'algorithme* **table3** *construit une table de multiplication pour tous les entiers de 1 à n en donnant tous leurs multiples jusqu'au n-ième. Il comprend deux boucles imbriquées, chacune effectuant n répétitions de son corps ; le corps de la boucle interne ne comporte qu'une multiplication. La complexité est ici $O(n^2)$.*

6.1.3 Différentes nuances de complexité

Complexité au pire

Pour deux données de même taille, un algorithme n'effectue pas nécessairement le même nombre d'opérations élémentaires. Par exemple, reprenons le test de primalité écrit au chapitre 4.

```
premier = True
for i in range(2, int(sqrt(n))+1):
    if n % i == 0:
        premier = False
        break
```

Si n est un nombre premier, alors il faudra $\sqrt{n} - 1$ itérations ; pourtant, pour $n - 1$ et $n + 1$ qui sont pairs, le programme s'arrête dès la première itération.

Si la fonction f caractérise l'efficacité d'un algorithme, on veut avoir l'assurance que l'exécution du programme sera terminée en un temps proportionnel à $f(n)$, éventuellement moins, mais pas plus. On cherche donc un majorant du temps d'exécution, autrement dit on retiendra le *pire des cas* pour exprimer la complexité.

Complexité dans le meilleur des cas

La complexité au pire est la plus significative, mais dans certains cas, il peut être utile de connaître aussi la complexité dans le meilleur des cas. Cette dernière reste d'un usage anecdotique mais elle peut donner une borne inférieure du temps d'exécution d'un algorithme.

En particulier, si la complexité dans le meilleur et dans le pire des cas sont du même ordre, cela signifie que le temps d'exécution de l'algorithme est relativement indépendant des données et ne dépend que de la taille du problème.

Complexité en espace

Jusqu'ici, on a uniquement discuté du temps d'exécution des algorithmes. Une autre ressource importante en informatique est la mémoire. On appelle *complexité en espace* d'un algorithme la place nécessaire en mémoire pour le faire fonctionner. Elle s'exprime également sous la forme d'un $O(f(n))$ où n est la taille du problème.

Évaluer la complexité en espace d'un algorithme ne pose la plupart du temps pas de difficulté ; il suffit de faire le total des tailles en mémoire des différentes variables utilisées. Une première exception à la règle est le cas où on alloue dynamiquement de l'espace mémoire au cours du programme (voir chapitre 12). L'autre cas est celui des fonctions récursives, qui cachent souvent une complexité en espace élevée (voir la section 5.3 pour un aperçu de l'empreinte mémoire d'une fonction récursive).

POUR ALLER PLUS LOIN Complexité en moyenne et complexité amortie

Pour un même algorithme, le temps d'exécution peut être très différent suivant les données d'entrée. C'est le cas par exemple de certains algorithmes triant un tableau de taille n, qui prennent un temps proportionnel à n si le tableau est trié et proportionnel à n^2 dans le pire cas. On s'intéresse donc parfois à la complexité en moyenne d'un algorithme. Parler de moyenne des temps d'exécution n'a de sens que si l'on a une idée de la fréquence des différentes données possibles pour un même problème de taille n. Les calculs de complexité moyenne recourent aux notions définies en mathématiques dans le cadre de la théorie des probabilités et des statistiques. Ils sont souvent très délicats et sortent du cadre de cet ouvrage.

Par ailleurs, il existe des problèmes où le pire cas peut se produire mais où, sur des exécutions répétées, on a la certitude qu'il ne se produira que peu fréquemment. On peut prendre l'exemple d'une personne désirant envoyer un SMS depuis un téléphone mobile. Quel est le coût en temps de cet envoi ? Si tout se passe bien, la rédaction et l'envoi se font en 2 minutes.

➡

En revanche, dans le pire cas, la batterie du téléphone est déchargée, il convient donc de la recharger pendant 4 heures. Le coût dans le pire cas pour un envoi est donc de 4 heures et 2 minutes.

Néanmoins, le coût pour n envois, où n est grand, est bien inférieur à $4n$ heures et $2n$ minutes. En effet, une fois le téléphone chargé, son utilisateur va pouvoir envoyer un millier de SMS avant de devoir recharger la batterie. On peut donc dire que dans le pire cas, l'envoi de n SMS successifs va demander $\left\lceil \frac{n}{1\,000} \right\rceil \times 4$ heures plus $2n$ minutes, On a donc la garantie que pour n grand, le temps mis pour envoyer n SMS est au plus de l'ordre de $n \times 14$ secondes plus $2n$ minutes. Autrement dit, le temps mis pour envoyer un SMS est de l'ordre de 2 minutes et 14 secondes. On dit que cette durée est la complexité *amortie* représentant le coût de l'envoi. La notion d'amortissement vient de la comptabilité : le coût d'un kilomètre en voiture est nul si la voiture fonctionne et si le plein est fait, alors qu'il est extrêmement élevé s'il faut commencer par acheter la voiture. La notion pertinente pour mesurer ce coût est en général de calculer l'amortissement des dépenses initiales (l'achat de la voiture) sur la totalité du kilométrage.

C'est une situation qu'on retrouve en informatique : il arrive ainsi que, dans certains problèmes, une opération ait un coût $O(n)$ dans le pire cas, où n est la taille du problème, et un coût constant en complexité amortie. En Python, l'opération d'ajout d'un nouvel élément à la fin d'un tableau de taille n rentre dans ce cadre.

Cependant, la théorie de la complexité amortie dépasse le cadre de cet ouvrage.

Exercice 6.3 Quelle est la complexité en temps d'un algorithme de division euclidienne procédant par soustractions successives ? Et sa complexité en espace ?

Comparer avec la complexité en temps et en espace de l'algorithme de division euclidienne que l'on apprend à l'école primaire et au collège (on ne cherchera pas à le programmer).

Exercice 6.4 Quelle est la complexité en temps de l'algorithme écrit dans l'exercice 4.36 ?

Exercice 6.5 * Quelle est la complexité en temps de la version récursive de l'algorithme de Horner présentée page 154 ?

Et sa complexité en espace ?

6.2 Structure de tableau

6.2.1 Construction d'un tableau

De manière simple, un *tableau*[2] n'est rien d'autre qu'une suite de valeurs stockées dans des cases mémoire contiguës. Ainsi, on représentera graphiquement le tableau contenant la suite de valeurs entières 3, 7, 42, 1, 4, 8, 12 de la manière suivante :

3	7	42	1	4	8	12

La particularité d'une structure de tableau est que le contenu de la i-ème case peut être lu ou modifié en temps constant, c'est-à-dire indépendant de i.

2. Dans la terminologie Python, la structure de données correspondante est appelée une « liste », ce qui est un peu malheureux. Nous donnerons un peu plus de détails sur cette structure dans le chapitre 12.

En Python, on peut construire un tel tableau en énumérant ses éléments entre crochets, séparés par des virgules :

```
In [1]: a = [3, 7, 42, 1, 4, 8, 12]
```

Les cases sont numérotées à partir de 0. On accède à la première case avec la notation `a[0]`, à la seconde avec `a[1]`, etc. Si on tente un accès en dehors des cases valides du tableau, alors on obtient une erreur :

```
In [2]: a[5]
Out[2]: 8
```

```
In [3]: a[7]
IndexError: list index out of range
```

Il est important de noter que `a[-1]`, ..., `a[-7]` ont du sens : `a[-1]` est le dernier élément (c'est-à-dire `12`), `a[-2]` l'avant-dernier (c'est-à-dire `8`), etc. On obtient la longueur d'un tableau avec la fonction `len`. Ainsi, `len(a)` vaut 7 sur l'exemple précédent. Un tableau peut avoir la longueur 0 ; il se note alors `[]`.

Il existe d'autres procédés pour construire un tableau que d'énumérer explicitement ses éléments. On peut par exemple concaténer un certain nombre de fois un tableau donné. Ainsi, l'expression suivante construit un tableau de taille 100 :

```
In [4]: 50 * [0,1]
Out[4]: [0, 1, 0, 1, ..., 0, 1]
```

Il est également possible d'utiliser une construction *par compréhension* de la forme `[e(i) for i in range(n)]`, où `e` est une expression quelconque qui construit le tableau `[e(0), e(1), ..., e(n-1)]`. Ainsi, l'expression suivante construit le même tableau que précédemment.

```
In [5]: [i % 2 for i in range(100)]
Out[5]: [0, 1, 0, 1, ..., 0, 1]
```

6.2.2 Accès aux éléments d'un tableau

On modifie le contenu d'une case avec la construction d'affectation habituelle. Ainsi, pour mettre à 0 la deuxième case du tableau `a`, on écrira :

```
In [6]: a[1] = 0
```

La fonction `print` affiche le contenu d'un tableau, sous la même forme que celle utilisée pour le construire :

```
In [7]: print(a)
[3, 0, 42, 1, 4, 8, 12]
```

On peut également extraire tous les éléments situés entre les indices i (inclus) et j (exclu) d'un tableau `t` avec la notation `t[i:j]`. Ainsi, l'instruction suivante :

```
In [8]: b = a[2:4]
```

affecte à la variable `b` un *nouveau* tableau de longueur 2 et de contenu `[42, 1]`. De manière générale, la notation `t[i:j]` construit un nouveau tableau de longueur $j - i$. Cette opération n'est pas instantanée, puisqu'elle revient à faire $j - i$ affectations. Il est important de comprendre que les cases mémoire de `a` ont été *copiées*. En particulier, aucune modification des cases de `b` ne modifiera le contenu de `a` (et réciproquement). Ainsi, la séquence d'instructions suivante affiche toujours `[3, 0, 42, 1, 4, 8, 12]`.

```
b[0] = 5
print(a)
```

En particulier, on peut obtenir une *copie* du tableau `a` avec `a[0:7]`, et plus généralement avec `a[0:len(a)]`. Il existe même un raccourci pour cela, à savoir `a[:]`.

ATTENTION Deux noms pour un même tableau

Si on écrit les deux instructions suivantes :

```
a = [0, 1, 2, 3]
b = a
```

alors le tableau `b` n'est pas une copie de `a`. En effet, les deux variables `a` et `b` désignent toutes les deux le même tableau. On dit alors que `b` et `a` sont des *alias*.

On peut s'en convaincre en modifiant un élément de `a` et en vérifiant que la modification s'observe également dans `b`. Si par exemple on exécute `b[2] = 42`, alors on obtient :

Cependant, `a` et `b` ne sont pas liés à jamais. Si on affecte à `a` un nouveau tableau, par exemple avec `a = [4, 5]`, alors on obtient la situation suivante :

où `b` désigne toujours le même tableau qu'auparavant.

POUR ALLER PLUS LOIN Les tableaux hétérogènes

En Python, les tableaux sont hétérogènes, c'est-à-dire que les valeurs de leurs composantes peuvent être de types différents, par exemple `[1, 'A', True]`. Dans cet ouvrage, on utilisera toujours des tableaux homogènes.

Comme on l'a vu au chapitre 3, les chaînes de caractères proposent les mêmes opérateurs que les tableaux pour accéder à un caractère, connaître la longueur de la chaîne, extraire une sous-chaîne, etc. La seule opération qui ne soit pas commune est la modification d'un caractère, les chaînes étant immuables. Une conséquence appréciable est que certains des programmes de ce chapitre sont utilisables sans changement sur des chaînes de caractères.

6.2.3 Parcours de tous les éléments d'un tableau

Pour exemple, on cherche à calculer la somme de tous les éléments d'un tableau d'entiers. Un algorithme simple pour cela consiste à initialiser une variable `s` à 0 et à parcourir tous les éléments du tableau pour les ajouter un par un à cette variable. La méthode naturelle pour effectuer ce parcours consiste à utiliser une boucle `for`. En effet, la construction `for i in range(n)` affecte successivement à la variable `i` les valeurs 0, 1, …, $n-1$. Ainsi peut-on écrire la fonction `somme` de la manière suivante :

```
def somme(a):
    s = 0
    for i in range(len(a)):
        s += a[i]
    return s
```

Cependant, on peut faire encore plus simple, car la boucle `for` de Python sait parcourir directement tous les *éléments* du tableau `a` avec `for x in a`. Ainsi, le programme se simplifie en :

```
def somme(a):
    s = 0
    for x in a:
        s += x
    return s
```

Ce programme effectue exactement `len(a)` additions, soit une complexité linéaire.
Comme exemple plus complexe, on considère l'évaluation d'un polynôme :

$$A(X) = \sum_{0 \leqslant i < n} a_i X^i$$

On suppose que les coefficients du polynôme A sont stockés dans un tableau `a`, le coefficient a_i étant stocké dans `a[i]`. Ainsi, le tableau `[1, 2, 3]` représente le polynôme $3X^2 + 2X + 1$. Une méthode simple, mais naïve, consiste à écrire une boucle qui réalise exactement la somme telle qu'elle est écrite ci-dessus. On a donc besoin d'accéder non seulement à l'indice `i`, mais aussi à la valeur `a[i]`. On peut parcourir le tableau en utilisant la construction `range`, comme plus haut :

```
def evaluer(a, x):
    s = 0
    for i in range(len(a)):
        s += a[i] * x**i
    return s
```

Là encore, il existe en Python une construction idiomatique, à savoir `for i, ai in enumerate(a)`, qui donne simultanément l'indice et la valeur de chaque case. Ainsi, on peut écrire :

```
def evaluer(a, x):
    s = 0
    for i, ai in enumerate(a):
        s += ai * x**i
    return s
```

Une méthode plus efficace pour évaluer un polynôme est d'utiliser la *méthode de Horner*. Elle consiste à réécrire la somme précédente de la manière suivante :

$$A(X) = a_0 + X(a_1 + X(a_2 + \cdots + X(a_{n-2} + Xa_{n-1})\dots))$$

Ainsi, on évite le calcul des différentes puissances X^i, en factorisant intelligemment et en ne faisant plus que des multiplications par X. Pour réaliser ce calcul, il faut parcourir le tableau de la droite vers la gauche, pour que le traitement de la `i`-ème case de `a` consiste à multiplier par X la somme courante, puis à lui ajouter `a[i]`. Si la variable `s` contient la somme courante, la situation est donc la suivante :

$$A(X) = a_0 + X(\cdots(a_i + X(\underbrace{a_{i+1} + \cdots}_{s})))$$

En Python, une manière de parcourir le tableau `a` de la droite vers la gauche consiste à utiliser la construction `for ai in reversed(a)`. Ainsi, la méthode de Horner s'écrit comme suit :

```
def horner(a, x):
    s = 0
    for ai in reversed(a):
        s = ai + x*s
    return s
```

On constate facilement que ce programme effectue exactement `len(a)` additions et autant de multiplications, soit encore une complexité linéaire.

ATTENTION Utiliser la récursivité avec précaution

On pourrait être tenté d'écrire une version récursive de la méthode de Horner de la façon suivante :

```
def horner_rec(p, x):
    if len(p) == 0:
        return 0
    else:
        return p[0] + x * horner_rec(p[1:], x)
```

Cette version évite notamment l'utilisation de `reversed`. Elle a cependant le défaut d'être plus longue à exécuter. En effet, l'expression `p[1:]` réalise une *copie* des éléments du tableau, ce qui revient à effectuer `len(p)-1` affectations. Au total, pour un tableau de taille initiale n, ce programme effectuerait donc $(n-1) + (n-2) + \cdots + 1$ affectations.

Exercice 6.6 *

1 Écrire une fonction qui prend un tableau d'entiers `t` en argument et renvoie le tableau des sommes cumulées croissantes correspondantes, autrement dit un tableau de même taille dont la k-ième composante vaut $\sum_{i=0}^{k}$ `t[i]`. Le tableau fourni en argument ne sera pas modifié.

2 Évaluer la complexité de cette fonction.

3 Est-il possible d'en écrire une version plus efficace ?

Exercice 6.7 Écrire une fonction qui renvoie un tableau contenant les n premières valeurs de la suite de Fibonacci (voir exercice 5.24).

6.3 Recherche dans un tableau

6.3.1 Recherche séquentielle

On cherche à déterminer si un tableau contient une certaine valeur. À la différence de la section précédente, on ne va pas nécessairement examiner *tous* les éléments du tableau, car on souhaite interrompre le parcours dès que l'élément est trouvé. Une solution consiste à utiliser une boucle `while`, de la façon suivante :

```
def appartient(x, a):
    i = 0
    while i < len(a) and a[i] != x:
        i += 1
    return i < len(a)
```

Il est important de noter que le caractère paresseux du `and` est ici crucial : il évite l'accès en dehors des bornes du tableau lorsque `i` atteint `len(a)`. On peut procéder autrement en utilisant la construction `return` à l'intérieur de la boucle pour interrompre son exécution. Du coup, on peut de nouveau utiliser une boucle `for` comme dans la section précédente :

```
def appartient(x, a):
    for y in a:
        if y == x:
            return True
    return False
```

On va maintenant écrire une fonction de recherche légèrement différente, qui renvoie le premier indice où la valeur `x` apparaît dans le tableau `s`. On utilise alors `enumerate` pour parcourir simultanément les indices et les valeurs correspondantes. Comme dans la fonction précédente, la construction `return` fait sortir de la fonction dès que la valeur `x` est trouvée. Si on sort de la boucle, on renvoie `None` pour signaler un échec de la recherche.

```
def indice(x, s):
    for i, y in enumerate(s):
        if y == x:
            return i
    return None
```

Une autre manière de signaler l'échec aurait consisté à utiliser une valeur entière non significative. Cependant, la solution avec `None` est plus robuste, par exemple car elle empêchera de traiter ce résultat comme un entier significatif dans un code qui appelle la fonction `indice`.

6.3.2 Recherche dichotomique dans un tableau trié

On note que dans le pire des cas, les fonctions `appartient` et `indice` précédentes parcourent tout le tableau et effectuent donc n comparaisons, où n est la longueur du tableau. Dans certains cas, cependant, la recherche d'un élément dans un tableau peut être réalisée de manière plus efficace. C'est le cas par exemple lorsque le tableau est trié. On peut alors exploiter l'idée suivante : on coupe le tableau en deux par le milieu et on détermine si la valeur `x` doit être recherchée dans la moitié gauche ou droite. En effet, il suffit pour cela de la comparer avec la valeur centrale. Puis, on répète le processus sur la portion sélectionnée.

On suppose par exemple que l'on cherche la valeur `9` dans le tableau `[1, 3, 5, 6, 9, 12, 14]`. La recherche s'effectue ainsi :

On cherche dans `a[0:7]`.	1	3	5	6	9	12	14
On compare `x`=9 avec `a[3]`=6.	1	3	5	6	9	12	14
On cherche dans `a[4:7]`.					9	12	14
On compare `x`=9 avec `a[5]`=12.					9	12	14
On cherche dans `a[4:4]`.					9		
On compare `x`=9 avec `a[4]`=9.					9		

Seules trois comparaisons ont été nécessaires pour trouver la valeur. C'est une application du principe *diviser pour régner*. On retrouvera d'autres applications de ce principe dans les chapitres 8 consacré à la résolution d'équations et 13 consacré aux tris.

Pour écrire l'algorithme, on délimite la portion du tableau `a` dans laquelle la valeur `x` doit être recherchée à l'aide de deux indices `g` et `d`. On maintient l'*invariant* suivant : les valeurs strictement à gauche de `g` sont inférieures à `x` et les valeurs strictement à droite de `d` supérieures à `x`, ce qui s'illustre ainsi :

0	`g`	`d` `n`
< `x`	?	> `x`

On commence par initialiser les variables `g` et `d` avec 0 et `len(a)-1`, respectivement :

```
def recherche_dichotomique(x, a):
    g, d = 0, len(a)-1
```

Tant que la portion à considérer contient au moins un élément :

```
    while g <= d:
```

on calcule l'indice de l'élément central, en faisant la moyenne de `g` et `d` :

```
        m = (g + d) // 2
```

Il est important de noter qu'on effectue ici une division *entière*. Qu'elle soit arrondie vers le bas ou vers le haut, on obtiendra toujours une valeur comprise entre `g` et `d`, ce qui assure d'une part que `a[m]` existe et qu'il est bien situé entre `g` et `d`. Si `a[m]` est l'élément recherché, on a terminé la recherche :

```
        if a[m] == x:
            return m
```

Sinon, on détermine si la recherche doit être poursuivie à gauche ou à droite. Si `a[m] < x`, on poursuit à droite :

```
        if a[m] < x:
            g = m+1
```

Sinon, on poursuit à gauche :

```
        else:
            d = m-1
```

Si on sort de la boucle `while`, c'est que l'élément ne se trouve pas dans le tableau, car il ne reste que des éléments strictement plus petits (à gauche de `g`) ou strictement plus grands (à droite de `d`). On renvoie alors `None` pour signaler l'échec.

```
    return None
```

Le code complet est donné programme 1 ci-dessous.

PROGRAMME 1 Recherche dichotomique dans un tableau trié

```
def recherche_dichotomique(x, a):
    """renvoie, si elle existe, la position d'une occurrence de x dans a
    supposé trié, et None sinon"""
    g, d = 0, len(a)-1
    while g <= d:
        m = (g + d) // 2
        if a[m] == x:
            return m
        if a[m] < x:
            g = m+1
        else:
            d = m-1
    return None
```

On veut maintenant montrer que la complexité de cet algorithme est au pire $O(\log n)$ où n est la longueur du tableau. En particulier, on effectue au pire un nombre logarithmique de comparaisons. La démonstration consiste à établir qu'après k itérations de la boucle, on a l'inégalité suivante :

$$d - g < \frac{n}{2^k}.$$

La démonstration se fait par récurrence sur k. Initialement, on a $g = 0$ et $d = n - 1$ et $k = 0$, donc l'inégalité est établie. On suppose maintenant l'inégalité vraie au rang k et $g \leqslant d$. À la fin de la $k+1$-ième itération, on a soit `g = m+1`, soit `d = m-1`. Dans le premier cas, on a donc :

$$d - \left(\left\lfloor \frac{g+d}{2} \right\rfloor + 1\right) \leqslant d - \frac{g+d}{2} = \frac{d-g}{2} < \frac{n}{2^k \times 2} = \frac{n}{2^{k+1}}.$$

Le second cas est laissé au lecteur. On conclut ainsi : pour $k \geqslant \log_2(n)$, on a $d - g < 1$, c'est-à-dire $d - g \leqslant 0$. On fait alors au plus une dernière itération.

La complexité de la recherche dichotomique est donc $O(\log n)$, alors que celle de la recherche séquentielle est $O(n)$. Il ne faut cependant pas oublier qu'elles ne s'appliquent pas dans les mêmes conditions : une recherche dichotomique est exclue si les données ne sont pas triées.

Exercice 6.8 Écrire une fonction qui renvoie l'élément maximal d'un tableau d'entiers. On discutera des diverses solutions possibles pour traiter le cas d'un tableau de longueur 0.

Exercice 6.9 * Écrire une fonction qui renvoie les deux plus grands éléments d'un tableau d'entiers. On supposera que le tableau est de longueur au moins 2 ; en revanche, on veillera à ne le parcourir qu'une seule fois.

Exercice 6.10

1. Écrire une fonction qui renvoie l'indice de la première occurrence de l'élément maximal d'un tableau d'entiers.
2. Évaluer la complexité de cette fonction.
3. Démontrer que tout algorithme répondant à cette question a une complexité au moins linéaire.

6.4 Recherche d'un mot dans un texte

Un problème classique en informatique consiste à rechercher, non pas une seule valeur, mais une *séquence* de valeurs dans un tableau. Cela revient à chercher une occurrence d'un tableau dans un autre ou, pour les chaînes de caractères, une occurrence d'un mot dans un texte. On souhaite donc écrire une fonction **recherche_mot** qui, étant donnés deux tableaux `m` et `t`, détermine la position de la première occurrence de `m` dans `t`, si elle existe, et qui renvoie **None** sinon. Ainsi, pour les tableaux `m=[1,2,3]` et `t=[2,1,4,1,2,6,1,2,3,7]`, **recherche_mot** renvoie 6 :

```
[2,1,4,1,2,6, 1,2,3, 7]
```

On effectue la recherche avec une boucle `for`, qui va considérer toutes les positions possibles pour le mot *m*, c'est-à-dire tous les indices *i* entre 0 et `len(t) - len(m)`, au sens large.

```
def recherche_mot(m, t):
    for i in range(1 + len(t) - len(m)):
```

On teste si le mot *m* apparaît à la position *i* avec une seconde boucle, qui compare les caractères de *m* et de *t* un à un. On utilise une variable *j* pour cela et on s'arrête, soit lorsque *j* atteint `len(m)`, soit lorsque les caractères diffèrent :

```
        j = 0
        while j < len(m) and m[j] == t[i + j]:
            j += 1
```

Il est important de noter que le caractère paresseux du `and` est encore ici crucial : il évite l'accès en dehors des bornes du tableau lorsque *j* atteint `len(m)`. Une fois sorti de la boucle `while`, on a reconnu le mot *m* à la position *i* si et seulement si `j == len(m)`, auquel cas on renvoie *i*. On interrompt ainsi l'exécution de la fonction dès la première occurrence trouvée :

```
        if j == len(m):
            return i
```

Sinon, on passe à la valeur suivante de *i*. Si on parvient à la fin de la boucle `for` principale, c'est qu'il n'y a pas d'occurrence de *m* dans *t*, ce que l'on signale en renvoyant `None` :

```
    return None
```

Le code complet est donné programme 2 ci-dessous.

Le pire des cas de cet algorithme correspond à la situation où on cherche sans succès le mot *m* à toutes les positions possibles dans *t* et où la boucle `while` parcourt néanmoins tous les caractères de *m*. C'est le cas par exemple lorsque l'on recherche le mot $X \ldots XY$ dans un texte constitué uniquement de X. La complexité dans le pire des cas est donc $|m| \times (|t| - |m| + 1)$. Dans le meilleur des cas, la complexité est clairement $|m|$.

PROGRAMME 2 Recherche d'un mot dans un texte

```
def recherche_mot(m, t):
    """renvoie, si elle existe, la position de la première occurrence du mot m
    dans le texte t et renvoie None sinon"""
    for i in range(1 + len(t) - len(m)):
        j = 0
        while j < len(m) and m[j] == t[i + j]:
            j += 1
        if j == len(m):
            return i
    return None
```

SAVOIR-FAIRE Concevoir un algorithme répondant à un problème précisément posé

1 Identifier la structure adaptée pour représenter les données du problème (par exemple un tableau).
2 Déterminer si le problème peut se ramener à un des algorithmes usuels sur cette structure (par exemple le parcours du tableau).
3 Apporter les modifications nécessaires à cet algorithme pour répondre au problème.

Exercice 6.11 avec corrigé Concevoir un algorithme vérifiant qu'une suite est croissante jusqu'à un certain rang.

1 On peut représenter les termes de la suite comme un tableau de flottants u.
2 Si la suite est effectivement croissante, il faudra le vérifier à chaque rang, mais si elle ne l'est pas, on pourra interrompre le parcours du tableau dès qu'on aura trouvé deux valeurs en ordre décroissant. L'algorithme que l'on va écrire est donc à rapprocher d'une recherche séquentielle.
3 Il y a principalement deux différences avec la recherche séquentielle. D'une part, on ne va pas comparer un élément avec une valeur fixée, mais avec l'élément suivant. D'autre part, on veut savoir si la suite est croissante et donc on renverra `True` *dans le cas où on a parcouru tout le tableau sans trouver de valeurs en ordre décroissant.*

On pourra écrire une fonction comme celle-ci :

```
def croissante(u):
    for i in range(len(u)-1):
        if u[i] > u[i+1]:
            return False
    return True
```

Exercice 6.12 Modifier le programme 2 pour qu'il affiche *toutes* les occurrences de `m` dans `t`. La complexité est-elle différente après cette modification ?

Exercice 6.13 Écrire une fonction qui vérifie qu'une chaîne de caractères est composée uniquement de lettres de l'alphabet, d'espaces et des symboles de ponctuation usuels.
Évaluer sa complexité.

Exercice 6.14 Écrire une fonction qui vérifie qu'une chaîne de caractères est une adresse e-mail valide. On pensera par exemple à vérifier la présence du symbole `@`, l'absence de certains caractères, etc.

Exercice 6.15 * Écrire une fonction qui vérifie qu'une chaîne de caractères est un palindrome, c'est-à-dire qu'elle est identique qu'on la lise de gauche à droite ou de droite à gauche.
Adapter cette fonction pour qu'elle ne tienne pas compte des espaces ni des signes de ponctuation.

6.5 Matrices

On peut choisir de représenter une matrice de dimensions (n, p) par un tableau de longueur n, dont les éléments sont des tableaux de longueur p.

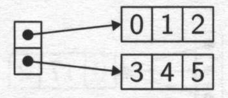
Ainsi, la matrice $(2, 3)$ suivante :

$$M = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

peut être définie en Python par le tableau *m* ci-après :

```
In [9]: m = [[0, 1, 2], [3, 4, 5]]
```

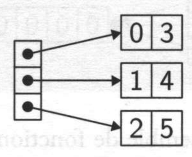
Cela correspond graphiquement à ceci :



En particulier, on accède à l'élément $M_{i,j}$ avec l'expression `m[i][j]`. Bien entendu, on aurait tout aussi bien pu faire le choix de représenter cette matrice par un tableau de longueur 3, dont les éléments auraient été les colonnes de la matrice :

```
In [10]: m = [[0, 3], [1, 4], [2, 5]]
```

Cela correspondrait graphiquement à ceci :



Cependant, il faudrait alors donner la colonne en premier pour accéder à un élément donné. Dans cet ouvrage, on adoptera toujours la première représentation, par proximité avec la notation $M_{i,j}$.

ATTENTION Partage de tableau

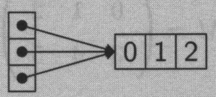
On cherche maintenant à construire la matrice suivante :

$$\begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$$

On pourrait être tenté de construire un tableau `v = [0, 1, 2]`, puis de l'utiliser trois fois pour chacune des lignes de la matrice :

```
v = [0, 1, 2]
m = [v, v, v]
```

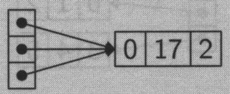
Il s'agit bien là d'une matrice de dimensions $(3, 3)$. Cependant, sa représentation en mémoire n'est pas la même que dans l'exemple précédent et montre au contraire un *partage* du tableau *v* entre les trois lignes.



En particulier, si on affecte un élément de la matrice, par exemple `m[0][1]` avec l'instruction suivante :

```
m[0][1] = 17
```

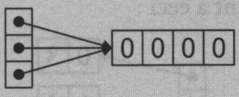
c'est en fait toute la colonne, c'est-à-dire les trois éléments `m[0][1]`, `m[1][1]`, `m[2][1]`, qui sont modifiés :



On peut l'observer facilement avec `print(m)`, qui affiche :

```
[[0, 17, 2], [0, 17, 2], [0, 17, 2]]
```

Pour la même raison, on ne peut pas utiliser l'expression `[[0] * 4] * 3` pour créer une matrice de dimensions $(4, 3)$ initialisée avec des zéros, car elle correspond en fait à la situation suivante :



On va présenter maintenant un ensemble de fonctions élémentaires sur les matrices. Le code complet est donné programme 3 page 166.

6.5.1 Création

Pour créer une matrice de grande taille, on ne souhaite évidemment pas donner tous ses éléments explicitement. Par ailleurs, les dimensions peuvent être contenues dans des variables. Écrivons donc une fonction `creer_matrice` pour construire une matrice M de dimensions (n, p) où chaque élément $M_{i,j}$ est initialisé avec une valeur *v*. On procède en créant un tableau de taille *n* initialisé avec `None` :

```
def creer_matrice(n, p, v):
    m = [None] * n
```

Puis on affecte à chacune de ses cases un tableau de taille *p* différent :

```
    for i in range(n):
        m[i] = [v] * p
    return m
```

Plus simplement, on peut utiliser la construction par compréhension pour faire la même chose :

```
def creer_matrice(n, p, v):
    return [[v] * p for i in range(n)]
```

On pourrait penser que, plus simplement encore, l'expression `[[v] * p] * n` suffit à construire cette matrice, mais ce n'est pas le cas, comme expliqué en détail page 161.

6.5.2 Copie

Si on veut copier une matrice `m` de dimensions (n, p), il faut prendre soin de copier chacune de ses n lignes, pour obtenir autant de nouveaux tableaux. De manière élémentaire, on commence par construire un tableau `r` de taille n initialisé à `None` :

```
def copie_matrice(m):
    n = len(m)
    r = [None] * n
```

Puis on affecte chaque ligne `r[i]` avec une *copie* de la ligne `m[i]` obtenue en utilisant la notation `m[i][:]` :

```
    for i in range(n):
        r[i] = m[i][:]
    return r
```

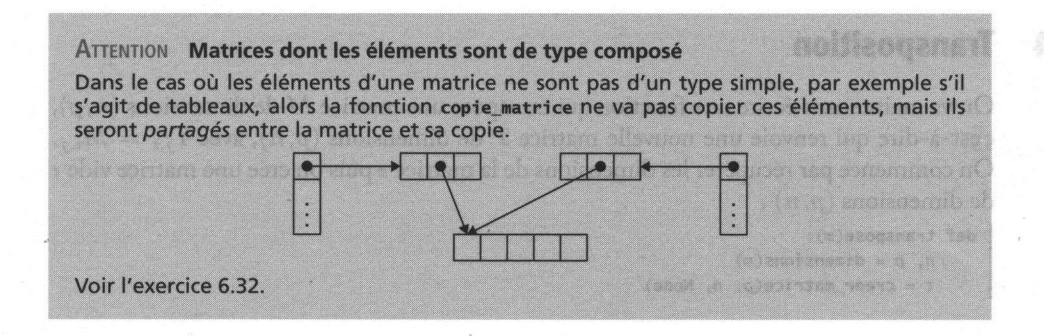
Là encore, on peut utiliser la notation par compréhension, pour réécrire plus simplement cette fonction :

```
def copie_matrice(m):
    return [m[i][:] for i in range(len(m))]
```

Quelle que soit la solution retenue, elle convient pour toutes les matrices dont les éléments sont des types simples (entiers, flottants, booléens). Plus généralement, elle convient pour toutes les matrices dont les éléments sont immuables, par exemple les chaînes de caractères.

ATTENTION Matrices dont les éléments sont de type composé

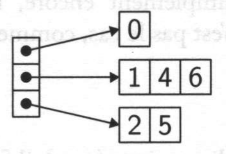
Dans le cas où les éléments d'une matrice ne sont pas d'un type simple, par exemple s'il s'agit de tableaux, alors la fonction `copie_matrice` ne va pas copier ces éléments, mais ils seront *partagés* entre la matrice et sa copie.



Voir l'exercice 6.32.

6.5.3 Dimensions

Un tableau de tableaux ne représente pas nécessairement une matrice de dimensions (n, p). En effet, rien n'empêche les tableaux qui représentent les lignes de la matrice d'être de longueurs différentes :



On va écrire une fonction `dimensions` qui vérifie qu'un tableau de tableaux `m` représente bien une matrice de dimensions (n, p), avec $n > 0$ et $p > 0$, et renvoie la paire (n, p). La fonction n'est pas définie (elle échouera) lorsque son argument ne représente pas une matrice. On commence par déterminer n comme la longueur du tableau `m` et par vérifier que $n > 0$:

```
def dimensions(m):
    n = len(m)
    assert n > 0
```

Pour déterminer la dimension p, il suffit de considérer la longueur de la première ligne `m[0]`. Cette ligne existe car $n > 0$. De même, on vérifie que $p > 0$:

```
    p = len(m[0])
    assert p > 0
```

Enfin, on vérifie que toutes les lignes de `m` ont bien la longueur p et on renvoie la paire (n, p) :

```
    for r in m:
        assert len(r) == p
    return (n, p)
```

Il est important de noter que la fonction `dimensions` fait l'hypothèse que son argument `m` est un tableau de tableaux. Dans le cas contraire, elle peut échouer.

6.5.4 Transposition

On va maintenant écrire une fonction qui transpose une matrice M de dimensions (n, p), c'est-à-dire qui renvoie une nouvelle matrice T de dimensions (p, n), avec $T_{j,i} = M_{i,j}$. On commence par récupérer les dimensions de la matrice `m` puis on crée une matrice vide `t` de dimensions (p, n) :

```
def transpose(m):
    n, p = dimensions(m)
    t = creer_matrice(p, n, None)
```

Puis on initialise les éléments de `t` avec une double boucle qui parcourt tous les éléments de `m` :

```
    for j in range(p):
        for i in range(n):
            t[j][i] = m[i][j]
    return t
```

Ici, on remplit `t` ligne par ligne, mais on aurait pu tout aussi bien la remplir colonne par colonne. Comme pour la création et la copie, on peut utiliser la notation par compréhension pour réécrire la fonction **transpose** de la manière suivante :

```
def transpose(m):
    n, p = dimensions(m)
    return [[m[i][j] for i in range(n)] for j in range(p)]
```

Bien que cette seconde version soit plus concise que la première, elle n'est pas forcément plus claire : en particulier, on ne voit pas facilement l'égalité $T_{j,i} = M_{i,j}$, alors qu'elle est manifeste dans la première version. De manière générale, nous n'abuserons pas de la notation par compréhension dans cet ouvrage.

6.5.5 Produit matriciel

On va terminer avec une opération courante, à savoir la multiplication de deux matrices `a` et `b`. On commence par récupérer les dimensions des deux matrices et par vérifier leur compatibilité, c'est-à-dire que le nombre de colonnes de `a` est égal au nombre de lignes de `b` :

```
def mult_matrice(a, b):
    n, p = dimensions(a)
    q, r = dimensions(b)
    assert q == p
```

On crée alors une nouvelle matrice `c` de dimensions (n, r), initialisée par 0, puis on effectue le calcul :

$$c_{i,j} = \sum_{k=0}^{p-1} a_{i,k} \times b_{k,j}$$

par une triple boucle sur les indices i, j et k :

```
    c = creer_matrice(n, r, 0)
    for i in range(n):
        for j in range(r):
            for k in range(p):
                c[i][j] += a[i][k] * b[k][j]
    return c
```

La structure de triple boucle de cette fonction montre clairement que sa complexité est $O(n \times p \times r)$. L'exercice 6.33 propose de réécrire cette fonction à l'aide de la transposition et du produit scalaire (sans en changer la complexité cependant).

Programme 3 Opérations élémentaires sur les matrices

```
def creer_matrice(n, p, v):
    """crée une nouvelle matrice de taille n x p,
    initialisée avec la valeur v"""
    return [[v] * p for i in range(n)]

def copie_matrice(m):
    """copie une matrice (mais pas ses éléments)"""
    return [m[i][:] for i in range(len(m))]

def dimensions(m):
    """vérifie que m est bien une matrice,
    et renvoie ses dimensions (lignes, colonnes)"""
    n = len(m)
    assert n > 0
    p = len(m[0])
    assert p > 0
    for r in m:
        assert len(r) == p
    return (n, p)

def transpose(m):
    """transpose une matrice"""
    n, p = dimensions(m)
    return [[m[i][j] for i in range(n)] for j in range(p)]

def mult_matrice(a, b):
    """multiplie deux matrices"""
    n, p = dimensions(a)
    q, r = dimensions(b)
    assert q == p
    c = creer_matrice(n, r, 0)
    for i in range(n):
        for j in range(r):
            for k in range(p):
                c[i][j] += a[i][k] * b[k][j]
    return c
```

6.6 Mode de passage des tableaux

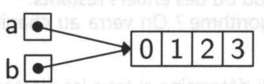
On considère le programme suivant, où une fonction `f` reçoit un tableau *b* en argument et le modifie :

```
def f(b):
    b[2] = 42
```

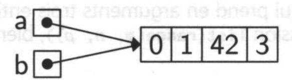
Le programme principal construit un tableau *a* et le passe à la fonction `f` :

```
a = [0, 1, 2, 3]
f(a)
```

Il est important de comprendre que, pendant l'exécution de la fonction `f`, la variable locale *b* est un alias pour le tableau *a* (voir page 152). Ainsi, à l'entrée de la fonction `f`, on a :



Après l'exécution de l'instruction `b[2] = 42`, on a :



En particulier, après l'appel à la fonction `f`, on a `a[2] == 42`.

Il est parfois utile d'écrire des fonctions qui modifient le contenu d'un tableau reçu en argument. Un exemple typique est celui d'une fonction qui échange le contenu de deux cases d'un tableau :

```
def echange(a, i, j):
    tmp = a[i]
    a[i] = a[j]
    a[j] = tmp
```

Elle peut aussi s'écrire plus simplement par affectation multiple :

```
def echange(a, i, j):
    a[i], a[j] = a[j], a[i]
```

Un autre cas est celui d'une fonction qui trie un tableau ; plusieurs exemples seront donnés dans le chapitre 13.

ATTENTION Les tableaux Python

Python propose plusieurs structures de tableaux, chacune prévue pour un usage spécifique. Dans ce chapitre, on a utilisé les « listes », dont on verra au chapitre 12 qu'elles sont en fait un peu plus que de simples tableaux.

Il existe également une bibliothèque appelée `array` qui, comme son nom l'indique, correspond effectivement à une structure de tableaux. Cependant, il s'agit uniquement d'une représentation plus compacte pour des tableaux très spécifiques, dont les éléments sont tous d'un même type numérique simple (caractère, entier ou flottant). En particulier, on ne pourrait pas utiliser la bibliothèque `array` pour représenter des matrices.

Enfin, la bibliothèque `numpy` propose aussi un type *ndarray* qui sert à représenter des tableaux de dimension arbitraire dont tous les éléments sont d'un même type, principalement utilisés pour le calcul matriciel. Attention à ne pas confondre ce type avec le précédent, même si ses valeurs s'affichent parfois aussi sous la forme `array(...)`.

6.7 Exercices

Exercice 6.16 * Le but de cet exercice est d'écrire une fonction qui détermine la médiane d'un tableau d'entiers.

1 Programmer l'algorithme suivant : on recherche le minimum et le maximum du tableau, on les supprime et on recommence jusqu'à obtenir un tableau de longueur inférieure ou égale à 2. On déduit alors facilement la valeur de la médiane du ou des entiers restants.
2 Quelle est la complexité de cet algorithme ? On verra au chapitre 13 qu'il existe des méthodes plus efficaces.

Exercice 6.17 Écrire une fonction qui détermine si tous les éléments d'un tableau (d'entiers ou de flottants) sont positifs ou nuls. On veillera à sortir de la fonction dès qu'une valeur négative est rencontrée.

Exercice 6.18 Écrire une fonction qui prend en arguments trois entiers m, n et p et renvoie un tableau identique à celui désigné par l'expression `list(range(m, n, p))`, bien entendu sans utiliser cette dernière expression.

Exercice 6.19 Écrire une fonction qui prend un entier n en argument et renvoie un tableau de n entiers tirés aléatoirement dans l'intervalle $[\![0, n]\!]$. On utilisera la fonction `random.randint` pour cela.

Exercice 6.20 Écrire une fonction qui renvoie un tableau aléatoire de caractères entre `'a'` et `'z'`. On utilisera pour cela les fonctions `ord` (qui renvoie le code d'un caractère) et `chr` (qui renvoie le caractère associé à un code donné).

Exercice 6.21 * Dans cet exercice, on écrira différentes versions d'une fonction qui prend en argument un tableau et inverse l'ordre de ses éléments, sans utiliser la fonction `reversed`.

1 Dans un premier temps, écrire une fonction qui renvoie un autre tableau contenant les éléments dans l'ordre inverse. Le tableau donné en argument devra rester inchangé.
2 Écrire ensuite une autre fonction qui renverse l'ordre des éléments en place, c'est-à-dire directement dans le tableau fourni en argument et sans utiliser de tableau auxiliaire.

Exercice 6.22 Écrire une fonction `appliquer` qui prend pour arguments une fonction `f` et un tableau `t` et renvoie le tableau formé des images de chaque élément de `t` par la fonction `f`.

La fonction `map` de Python réalise une opération similaire.

Exercice 6.23 * Écrire une fonction qui vérifie si un tableau donné en argument est dépourvu de doublons (aucune valeur n'y apparaît deux fois). Évaluer sa complexité.

Exercice 6.24 Écrire une fonction qui mélange aléatoirement les éléments d'un tableau `a` en utilisant le « mélange de Knuth » (*Knuth shuffle*). L'algorithme est le suivant : pour `i` allant de 1 à `len(a)-1`, on échange `a[i]` avec `a[j]`, où `j` est un entier choisi aléatoirement entre `0` et `i`, inclus.

Exercice 6.25 Pour tester la fonction `recherche_dichotomique` (programme 1 page 157), écrire une fonction qui renvoie un tableau trié de n entiers. On pourra procéder en utilisant une variable entière, que l'on incrémentera de manière aléatoire.

Exercice 6.26 * Réécrire la fonction `recherche_mot` (programme 2 page 159), en utilisant la construction `t[i:j]`. Quel est le défaut de cette solution ?

Exercice 6.27 Écrire une fonction qui calcule le produit scalaire de deux vecteurs représentés par deux tableaux de même taille.

Exercice 6.28 En s'inspirant de la fonction **`creer_matrice`**, écrire les trois fonctions suivantes :

1 une fonction qui prend un entier $n > 0$ en argument et renvoie la matrice identité de dimension n ;

2 une fonction qui prend un entier $n > 1$ en argument et renvoie la matrice tridiagonale de dimension n suivante :

$$\begin{pmatrix} 1 & 1 & & & (0) \\ 1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ (0) & & & 1 & 1 \end{pmatrix}$$

3 une fonction qui prend un entier $n > 0$ en argument et renvoie la matrice de dimension n formée sur le modèle suivant, ici pour $n = 4$:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 2 & 3 \\ 3 & 2 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

Exercice 6.29 * Écrire une fonction qui décide si une matrice donnée est symétrique et une autre qui décide si elle est antisymétrique. Quelle est la complexité de ces fonctions ?

Exercice 6.30

1 Définir en Python deux matrices `a` et `b` représentant respectivement :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

2 Quelle est la valeur de l'expression `a + b` ? Pourquoi ?

3 Écrire une fonction **`somme_matrices`** qui calcule la somme de deux matrices de mêmes dimensions.

4 Quelle est la valeur d'une expression comme `3 * a` ? Pourquoi ?

5 Écrire une fonction **`scal_matrice`** qui prend en argument une matrice (de dimensions quelconques) et un flottant, et calcule le produit de cette matrice par ce flottant.

Exercice 6.31 ** Dans cet exercice, on programme différents jeux d'alignements dans une grille.

1 Écrire un programme qui permet à deux joueurs humains de jouer au Tic-tac-toe. Le programme vérifiera que les coups joués sont valides et détectera la victoire d'un joueur ou la partie nulle.

2 Modifier ce programme pour qu'il joue de façon aléatoire contre un joueur humain.

3 Modifier ce programme pour qu'il joue de façon suffisamment stratégique pour ne jamais perdre, qu'il soit premier ou second à jouer.

4 Écrire un programme qui permet à deux joueurs humains de jouer au Morpion : la grille de jeu est cette fois-ci rectangulaire, de dimensions choisies en début de partie et il faut réaliser un alignement de 5 pions pour gagner.

Exercice 6.32 * Écrire une variante de la fonction **`copie_matrice`** qui copie une matrice dont les éléments sont des tableaux, de manière à ce que les éléments ne soient pas partagés entre la matrice et sa copie. (Note : il existe en Python une fonction de bibliothèque **`copy.deepcopy`** qui effectue une copie récursive des valeurs.)

Exercice 6.33 Écrire une fonction qui multiplie deux matrices en utilisant la fonction **`transpose`** et le produit scalaire (voir exercice 6.27 ci-dessus).

Exercice 6.34 * Écrire une fonction qui élève une matrice carrée à la puissance n en réutilisant l'idée de l'exercice 4.25. Quelle est sa complexité ?

Exercice 6.35 * Déduire de l'exercice précédent une fonction qui calcule le n-ième élément de la suite de Fibonacci (voir exercice 5.24) avec $\log n$ produits matriciels, en utilisant l'identité suivante :

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Exercice 6.36 Écrire une fonction **`construire_matrice(n, p, f)`** qui renvoie la matrice de dimensions (n, p) dont le terme général est `f(n, p)`.

Troisième partie

Ingénierie numérique et simulation

Dans cette partie, nous nous intéressons à des méthodes numériques pour la résolution de systèmes linéaires (chapitre 7), d'équations sur les réels (chapitre 8) et d'équations différentielles (chapitre 9). D'une part, nous rappelons les méthodes du cours de mathématiques, qui sont faciles à programmer, et dont la validité peut être démontrée formellement et sans grande difficulté. D'autre part, nous expliquons comment utiliser les fonctions « clés en main » fournies par Python et ses bibliothèques adaptées, avec des exemples rencontrés en mathématiques, sciences physiques ou chimie.

L'idée pénible à garder à l'esprit est que, *en calcul numérique, tout est faux !* En effet, les données prises en entrée sont des approximations des données « réelles » (parce qu'issues d'autres calculs ou de mesures physiques). De plus, on résout des équations qui sont une approximation de la vie réelle (linéarisation d'un phénomène en physique...) et on applique pour cela des schémas qui introduisent une erreur dans le résultat. Enfin, le moindre calcul en arithmétique flottante induit des erreurs d'arrondis, on note le résultat final sur un morceau de papier, et on se trompe en copiant la deuxième décimale.

Nous montrons sous quelles conditions on peut contrôler ces erreurs, et comment choisir les paramètres des méthodes utilisées pour obtenir un résultat satisfaisant.

7

Pivot de Gauss et résolution de systèmes

Le pivot de Gauss est un outil crucial en algèbre linéaire — du point de vue théorique comme dans les aspects algorithmiques. Les situations qui s'y réduisent sont en effet omniprésentes et la méthode du pivot proprement dite a la double particularité de bien s'exprimer sous forme d'algorithme, mais aussi de mettre en exergue les problèmes de représentation des nombres en machine. Nous en expliquons dans ce chapitre le principe et la mise en œuvre.

La résolution d'un système linéaire est une activité formatrice pour un étudiant commençant des études scientifiques. D'une part, c'est l'occasion de réfléchir à des notions clés telles que les *équivalences* (leur validité, leur pertinence), la *description* d'un ensemble de solutions, et même la notion d'*équation* : que signifie « *résoudre* $ax = b$ » ? D'autre part, il s'agit de perdre la mauvaise habitude fréquente consistant à « bricoler » les équations pour arriver à une solution [1] dont on prétend avec plus ou moins de conviction que c'est *la* solution.

L'algorithme du pivot de Gauss sert à *résoudre* un système linéaire au sens où, partant d'un système à n équations et p inconnues, il va fournir un système *équivalent* permettant de paramétrer l'ensemble des solutions (s'il est non vide), ou de *démontrer* qu'il n'y a pas de solution en fournissant une condition nécessaire non compatible. Point important d'un tel algorithme : il ne laisse aucune place à l'astuce, on se contente d'exécuter des tâches répétitives mais simples, comme lorsqu'on a appris à additionner ou multiplier des entiers en primaire. L'expérience montre que le plus difficile reste d'accepter de changer de « méthode », pour peu qu'on puisse nommer ainsi ce qu'on pratiquait en général face à un système linéaire !

L'algorithme du pivot de Gauss, essentiellement basé sur des transvections (opérations sur les lignes, de la forme $L_j \leftarrow L_j - \lambda L_i$) est très simple à mettre en œuvre, du moins lorsqu'il n'y a pas de paramètres formels, et il possède une complexité raisonnable, à savoir cubique en la taille de la matrice.

Il existe de nombreuses variations autour du pivot, y compris pour la résolution de systèmes linéaires, mais on verra que les mêmes idées se retrouvent dans des problèmes tels que :

- inverser une matrice (inversible) ;
- déterminer deux matrices triangulaires L et U (respectivement inférieure et supérieure) telles que $A \in \mathcal{M}_n(\mathbb{R})$ s'écrive LU (ou $LT_\sigma U$ avec T_σ une matrice de permutation — décomposition de *Bruhat*) ;
- calculer le déterminant de $A \in \mathcal{M}_n(\mathbb{R})$;
- calculer le rang de $A \in \mathcal{M}_{n,p}(\mathbb{R})$;
- déterminer deux matrices inversibles P et Q telles que $PAQ = J_r$, avec r le rang de A ;
- etc.

Nous serons confrontés principalement à deux problèmes :

- **La précision du résultat** : elle dépend évidemment de celle des données, mais même avec des données exactes, les erreurs d'arrondis peuvent induire des erreurs dans le résultat, plus ou moins importantes selon la méthode choisie.
- **La comparaison d'un réel à zéro** : les calculs avec les flottants induisent des erreurs, qui peuvent faire apparaître ou au contraire disparaître le réel nul (voir chapitre 2). Comparer un coefficient à zéro n'a donc pas grand sens, alors que dans l'algorithme du pivot de Gauss, il est crucial de s'assurer que le pivot en est bien un, c'est-à-dire qu'il est non nul !

1. En fait, plutôt un « candidat-solution », les bricolages fournissant des conditions nécessaires sur les inconnues... conditions dont on ne sait pas si elles sont suffisantes pour que les équations initiales soient vérifiées.

7.1 Résolution de AX = Y : principe du pivot

L'algorithme du pivot de Gauss sait résoudre des systèmes généraux à n équations et p inconnues. Pour simplifier, on se placera pour l'essentiel de l'exposé dans le cas où $n = p$, avec l'hypothèse supplémentaire de l'existence d'une et une seule solution. On parle de *système de Cramer*. Dans les exercices, on s'autorisera à sortir un peu de ce cadre.

Exercice 7.1 Proposer un système à deux équations et deux inconnues ne possédant aucune solution, puis un autre possédant une infinité de solutions.

7.1.1 Le cas des systèmes triangulaires

Les systèmes triangulaires sont très simples à résoudre.
Exemple 1.

$$\left\{\begin{array}{rcl} 2x + 2y - 3z & = & 2 \\ y - 6z & = & -3 \\ z & = & 4 \end{array}\right. \iff \left\{\begin{array}{rcl} 2x + 2y - 3z & = & 2 \\ y & = & 6z - 3 = 21 \\ z & = & 4 \end{array}\right.$$

$$\iff \left\{\begin{array}{rcl} x & = & \frac{1}{2}(-2y + 3z + 2) = -14 \\ y & = & 21 \\ z & = & 4 \end{array}\right.$$

Dès maintenant, le lecteur doit bien comprendre le rôle de l'équivalence, qui assure que le système initial possède une unique solution. Les exercices 7.2 et 7.3 précisent la pertinence puis la validité de ces équivalences.

Exercice 7.2 Dans les équivalences précédentes, quelles sont les implications qui assurent l'existence d'une solution ? Lesquelles donnent l'unicité ?

Exercice 7.3 Expliquer pourquoi, lors des substitutions, on a bien gardé des systèmes équivalents entre eux.

La résolution d'un système linéaire passera systématiquement par une première étape pour le mettre sous forme triangulaire en gardant l'équivalence avec le système initial. Si une équation « disparaît » ou bien fournit une contradiction, alors on peut en conclure que :

- Si l'équation $0 = 0$ apparaît, on peut l'éliminer du système *en gardant l'équivalence*.
- Si l'équation $0 = \beta$ apparaît (avec $\beta \neq 0$), le système initial n'admet pas de solution (sens $\Rightarrow$ de l'équivalence).

Pour un système de Cramer, ces situations ne se présenteront pas.

La résolution d'un système triangulaire sera la dernière étape (la plus simple et rapide) de la résolution d'un système linéaire. On l'appelle souvent *phase de remontée* : on résout les

équations de bas en haut, en substituant aux inconnues les valeurs trouvées dans les lignes inférieures.

ATTENTION Systèmes paramétrés
On peut être confronté à des systèmes faisant intervenir des paramètres : si on rencontre l'équation $0 = \beta$ où β est un paramètre (ou une combinaison de paramètres), il faut alors commencer une discussion, qui aboutit en général à différentes conclusions. Ne serait-ce que pour une équation aussi simple que $ax = b$, l'inconnue étant x, on peut être amené à discuter sur $a = 0$. Dans le cadre d'une résolution *numérique* de système linéaire, on ne rencontrera pas cette situation.

7.1.2 Les transvections

Pour mettre un système sous forme triangulaire, on va réaliser des *transvections*, c'est-à-dire des opérations élémentaires de la forme suivante : « ajouter tant de fois telle équation à telle autre », afin d'éliminer[2] des inconnues dans des équations... mais pas dans le désordre !

On va décrire ces opérations pour un système $(3, 3)$ d'inconnues (x, y, z) dans une situation favorable, en notant L_1, L_2 et L_3 les trois équations en jeu.

On suppose que le coefficient en x de la première équation, appelé a, est non nul. On va s'en servir comme *pivot* pour éliminer les autres occurrences de x. Si on note b et c les coefficients en x des deuxième et troisième lignes, le système constitué des équations L_1, $L'_2 = L_2 - \frac{b}{a}L_1$ et $L'_3 = L_3 - \frac{c}{a}L_1$ est alors équivalent au premier (le démontrer n'est pas inutile) et ne fait apparaître x que dans la première équation. En supposant que le coefficient en y de la nouvelle deuxième ligne L'_2, appelé d, est non nul (c'est alors le nouveau pivot) et en notant e celui de y dans la nouvelle troisième ligne L'_3, le système constitué des lignes L_1, L'_2 et $L'_3 - \frac{e}{d}L'_2$ est équivalent au premier système et triangulaire : on est ramené à un cas qu'on sait traiter.

Lors de la première étape, on ne touche pas à la première ligne. De même, à la deuxième étape, on ne touche ni à la première ni à la deuxième ligne, et ainsi de suite.

Dans l'exemple qui suit, on adopte une notation classique : pour dire qu'on change la seconde ligne L_2 en $L'_2 = L_2 + \alpha L_1$, on préférera noter $L_2 \leftarrow L_2 + \alpha L_1$. Cela signifie qu'on appelle désormais L_2 ce que désignait auparavant $L_2 + \alpha L_1$. Après cinq opérations de ce type, on parlera donc toujours de L_8 plutôt que de L_8'''''.

2. Les anglo-saxons parlent de *Gaussian Elimination*.

Exemple 2.

$$\begin{cases} \mathbf{2}x + 2y - 3z = 2 \\ -2x - y - 3z = -5 \\ 6x + 4y + 4z = 16 \end{cases} \underset{L_3 \leftarrow L_3 - 3L_1}{\overset{L_2 \leftarrow L_2 + L_1}{\Longleftrightarrow}} \begin{cases} 2x + 2y - 3z = 2 \\ \mathbf{1}y - 6z = -3 \\ -2y + 13z = 10 \end{cases}$$

$$\overset{L_3 \leftarrow L_3 + 2L_2}{\Longleftrightarrow} \begin{cases} 2x + 2y - 3z = 2 \\ y - 6z = -3 \\ z = 4 \end{cases} \Longleftrightarrow \ldots$$

Dans les cas moins favorables, on peut rencontrer en cours de résolution d'un système $(3,3)$ ces trois problèmes :

- **Le pivot n'est pas là où on veut** : si, à la première étape, le coefficient en x de la première ligne est nul, on peut échanger la première équation avec la deuxième ou la troisième. De même, si à la seconde étape, le coefficient en y (futur pivot) est nul, on peut échanger la deuxième équation avec la troisième, *mais pas la première* (se souvenir qu'on veut arriver à un système triangulaire : il ne faut pas faire réapparaître x dans les deux dernières équations).
- **Il n'y a plus de pivot en une variable** : si tous les coefficients en x sont nuls (c'est rare : cela revient à dire que x n'apparaît pas dans le système...), on peut prendre y ou z comme première variable. De même, si après la première étape, y n'apparaît ni dans la deuxième ni dans la troisième équation, on peut prendre z comme deuxième inconnue.
- **Il n'y a plus de pivot** : cela signifie que les membres de gauche des équations restantes sont nuls. Selon que les membres de droite correspondants sont nuls ou pas, ces équations vont disparaître ou bien rendre le système incompatible.

Les deux dernières situations ne se produiront pas sur un système de Cramer.

7.1.3 Le problème de la comparaison à zéro

Lorsqu'on travaille avec des systèmes à coefficients rationnels et que les calculs se font avec des objets représentant effectivement des fractions (un couple d'entiers, ce qui ne pose pas de problème en Python), on manipule en permanence des objets informatiques représentant de façon exacte les objets mathématiques associés. C'est également le cas dans d'autres cadres tels que les corps finis : les objets manipulés sont plus élaborés que de simples entiers ou couples d'entiers, mais il n'y a pas de *perte d'information* dans leur représentation informatique, puis leur manipulation.

Avec des flottants, il n'en va pas de même, comme on l'a vu au chapitre 2.

Exercice 7.4 avec corrigé Que vaut $12\left(\frac{1}{3}-\frac{1}{4}\right)-1$? Et qu'en pense Python ?

```
In [1]: 12*(1./3-1./4)-1
Out[1]: -2.220446049250313e-16
```

En pratique :

- Les calculs approchés peuvent faire apparaître des termes petits, mais non nuls... censés pourtant représenter le réel nul. Ceci va transformer en système de Cramer des systèmes qui n'en étaient pas, ou conduire (même pour un système de Cramer) à prendre comme pivot une quantité très faible issue de l'accumulation d'approximations. Même pour un système de Cramer, le résultat annoncé n'aura alors plus aucun sens [3].
- Ces termes infinitésimaux risquent en outre d'être utilisés comme pivots : on prendra leur inverse, ce qui produira de grands flottants, eux-mêmes évidemment sans signification.
- *A contrario*, on peut voir apparaître des coefficients nuls, alors que les objets mathématiques qu'ils représentent ne le sont pas.

Ainsi, le système suivant n'a pas de solution :

$$\begin{cases} x + 1/4y + z = 0 \\ x + 1/3y + 2z = 0 \\ \quad\quad y + 12z = 1 \end{cases}$$

Pourtant, sa résolution numérique fournira un résultat, présenté à tort comme une solution. De même, la matrice $\begin{pmatrix} 1 & 1/4 & 1 \\ 1 & 1/3 & 2 \\ 0 & 1 & 12 \end{pmatrix}$ est de rang 2, mais son calcul numérique par pivot risque de donner 3 comme résultat :

```
In [2]: resolution([[1,1./4,1],[1,1./3,2],[0,1,12]],[[0],[0],[1]])
Out[2]: [-750599937895082.8, 4503599627370496.0, -375299968947541.25]
```

Inversement, le système suivant, lui, est de Cramer :

$$\begin{cases} x + (10^{15}+1)y + z = 1 \\ x + (1+10^{-15})y + 2z = 0 \\ \quad\quad 10^{15}y + z = 0 \end{cases}$$

3. La méthode du pivot partiel permettra néanmoins de pallier ce problème.

Pourtant, lors de sa résolution numérique par pivot de Gauss, après deux transvections, la troisième équation aura son membre de gauche nul. De même, la matrice $\begin{pmatrix} 1 & 1+10^{15} & 1 \\ 1 & 1+10^{-15} & 2 \\ 0 & 10^{15} & 1 \end{pmatrix}$ est de rang 3, mais son calcul numérique par pivot risque de donner 2 comme résultat :

```
In [3]: rank(array([[1,1+10**15,1],[1,1+10**(-15),2],[0,10**15,1]]))
Out[3]: 2
```

On retrouve ainsi le simple exemple de l'exercice 7.4 : tester la nullité d'un flottant (ou l'égalité de deux flottants) est fatalement trompeur et dangereux.

ATTENTION L'égalité et les flottants

Tester l'égalité de deux flottants n'a presque jamais de sens, est toujours risqué et doit donc être évité... sauf bonnes raisons, qui sont alors à expliciter !

Pour le problème auquel on s'intéresse, les tests de nullité sont effectués pour trouver un pivot. Si on sait que le système est de Cramer, on peut chercher sur la colonne en cours le coefficient le plus élevé en valeur absolue (ou module). Cela n'interdit pas les mauvaises surprises [4], mais cela marche en pratique plutôt bien. Cela s'appelle la méthode du *pivot partiel*, qu'on va exposer dans la section suivante.

7.1.4 Formalisation de l'algorithme

On fait ici l'hypothèse que le système initial est de Cramer. Les autres cas seront décrits en cours de mathématiques. Il est important de noter que les opérations réalisées vont introduire des systèmes équivalents au premier, qui demeureront donc des systèmes de Cramer.

Comme signalé plus haut, on veut éliminer des variables dans les équations successives. On va donc faire en sorte qu'après k étapes, pour tout i entre 1 et k, la i-ème variable ait disparu de toutes les équations du système à partir de la $(i+1)$-ème. Ce sera l'invariant de boucle.

Ainsi, après la $(n-1)$-ème étape, le système sera bien sous forme triangulaire.

4. On peut toujours construire des contre-exemples mettant en déroute cette technique.

Dans le pseudo-code qui suit, on résout le système $Ax = y$. La ligne L_i désigne à la fois les coefficients de A (qui sont dans une matrice, un tableau bidimensionnel) et les seconds membres, qui sont dans une matrice colonne y. Les indexations de tableaux vont de 0 à $n-1$ comme en Python :

pour i *de* 0 *à* $n-2$ **faire**
  Trouver j entre i et $n-1$ tel que $|a_{j,i}|$ soit maximale.
  Échanger L_i et L_j (coefficients de la matrice **et** membres de droite).
  pour k *de* $i+1$ *à* $n-1$ **faire**
    $L_k \leftarrow L_k - \dfrac{a_{k,i}}{a_{i,i}} Li$

Rechercher j entre i et n tel que $|a_{j,i}|$ soit maximale (puis échanger deux lignes) a deux objectifs : d'une part s'assurer que le coefficient en position (i,i) sera différent de 0 (c'est essentiel pour pouvoir pivoter) et, d'autre part, minimiser les erreurs numériques dans la suite du calcul.

Arrivé ici, le système est sous forme triangulaire et il n'y a plus qu'à « remonter », via des substitutions. Le résultat est mis dans un tableau x et il s'agit donc de calculer :

$$x_i = \frac{1}{a_{i,i}}\left(y_i - \sum_{k=i+1}^{n-1} a_{i,k}x_k\right)$$

pour i *de* $n-1$ *à* 0 **faire**
  pour k *de* $i+1$ *à* $n-1$ **faire**
    $y_i \leftarrow y_i - a_{i,k}x_k$
  $x_i \leftarrow \dfrac{y_i}{a_{i,i}}$

Exercice 7.5 Montrer que le caractère « de Cramer » d'un système ne dépend pas des membres de droite des équations.

7.1.5 Le formalisme matriciel

Décrivons rapidement le pivot de Gauss dans le cadre matriciel. On note que ce point de vue peut être mis de côté dans un premier temps si le cours sur les matrices n'a pas encore été traité en mathématiques.

Un système linéaire peut être vu comme une équation matricielle : le système de l'exemple 2 se traduit par $AX = Y$, avec $A = \begin{pmatrix} 2 & 2 & -3 \\ -2 & -1 & -3 \\ 6 & 4 & 4 \end{pmatrix}$, $X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ et $Y = \begin{pmatrix} 2 \\ -5 \\ 16 \end{pmatrix}$.

Il est équivalent au système $UX = Y'$, avec $U = \begin{pmatrix} 2 & 2 & -3 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{pmatrix}$ et $Y' = \begin{pmatrix} 2 \\ -3 \\ 4 \end{pmatrix}$.

Lorsque $i \neq j$, faire l'opération élémentaire $L_i \leftarrow L_i + \lambda L_j$ revient à multiplier (à gauche) la matrice A par la matrice de transvection $T_{i,j}(\lambda)$ de terme général :

$$t_{k,l} = \begin{cases} 1 & \text{si } k = l, \\ \lambda & \text{si } k = i \text{ et } l = j, \\ 0 & \text{sinon.} \end{cases}$$

De même, des transvections sur les colonnes (inutiles pour les systèmes de Cramer) correspondraient à des multiplications à droite par des matrices de transvection.

Dans ce cadre, on peut interpréter ainsi la résolution de $AX = Y$: on a trouvé N matrices de transvection (ou d'échange) $T_1, ..., T_N$ telles que $AX = Y$ soit équivalent à $\underbrace{T_N T_{N-1} ... T_1}_{T} AX = Y'$. Les substitutions de la phase de remontée s'interprètent à nouveau comme des opérations sur les lignes, de sorte qu'on arrive à un système de la forme $\underbrace{S_M S_{M-1} ... S_1}_{S} TAX = Y''$, avec $STA = I_n$. Ainsi, le système initial $AX = Y$ est équivalent à $X = Y''$: on a bien résolu le système.

Si enfin on s'intéresse à A supposée inversible sans chercher à résoudre $AX = Y$, on adapte facilement la méthode de résolution de cette dernière pour *expliciter* T inversible telle que TA soit triangulaire supérieure, puis S telle que $STA = I_n$: la matrice ST est alors l'inverse de A.

Exercice 7.6 Vérifier que l'opération élémentaire $L_i \leftarrow L_i + \lambda L_j$ revient à multiplier A à gauche par $T_{i,j}(\lambda)$. (Presque) sans calcul, démontrer que $T_{i,j}(\lambda)$ est inversible et déterminer son inverse.

Exercice 7.7 Interpréter les échanges de lignes $L_i \leftrightarrow L_j$ et les dilatations $L_i \leftarrow \lambda L_i$ (avec $\lambda \neq 0$) comme des multiplications par des matrices particulières.

POUR ALLER PLUS LOIN Presque toutes les matrices ont une décomposition LU.

Tout d'abord, si on prend des coefficients aléatoires (en différents sens raisonnables), on obtient avec probabilité 1 un système de Cramer (une matrice inversible).

Encore mieux : on n'aura en général pas de problème de pivot ! Plus précisément, si les *mineurs principaux* (les matrices (k,k) « en haut à gauche » extraites de la matrice initiale) sont tous inversibles, alors à chaque étape k du pivot[5], le coefficient présent en position (k,k) est non nul, et on peut l'utiliser pour pivoter.

Ainsi, partant d'une matrice A, on trouve une matrice L_1 triangulaire inférieure, telle que $U = L_1 A$ soit triangulaire supérieure. Si on note $L = L_1^{-1}$, on a la décomposition $A = LU$ (L pour *Lower* et U pour *Upper*).

Si par malheur on trouve à l'étape k le coefficient $a_{k,k}$ qui est nul, alors on prend le plus petit $j > k$ tel que $a_{j,k}$ soit non nul et on l'utilise comme pivot pour placer des zéros dessous et à droite par des transvections sur les colonnes. On parvient ainsi, modulo quelques dernières dilatations, à construire deux matrices L et U respectivement triangulaires inférieure et supérieure, telles que $A = LT_\sigma U$, avec T_σ une matrice de permutation : c'est la *décomposition de Bruhat*[6].

Les matrices L et U ne sont pas uniques, mais la permutation σ l'est. Dans la partition de $GL_n(\mathbb{R})$ en $n!$ composantes, celles associées à l'identité — c'est-à-dire celles possédant une décomposition LU — constituent la « grosse cellule » : c'est un ouvert dense.

7.2 Mise en œuvre

7.2.1 Découper le travail

On commence par réfléchir aux bons outils, à savoir les fonctions et programmes auxiliaires à l'aide desquels l'écriture du programme principal sera quasiment une traduction en anglais de l'algorithme ! Il faudra déléguer les opérations suivantes :

- la recherche d'un pivot ;
- les échanges de lignes ;
- les transvections.

Le premier programme prend en entrée une matrice A et un indice i. Il doit renvoyer un indice $j \geqslant i$ tel que $|a_{j,i}|$ soit maximale :

```
def chercher_pivot(A, i):
    n = len(A) # le nombre de lignes
    j = i # la ligne du maximum provisoire
    for k in range(i+1, n):
        if abs(A[k][i]) > abs(A[j][i]):
            j = k # un nouveau maximum provisoire
    return j # en faisant bien attention à l'indentation :-)
```

5. Le pivot normal et non partiel.
6. Les analystes numériciens ont tendance à privilégier l'ordre LUP (la matrice de permutation en dernier).

Les échanges de lignes se passent de commentaires :

```
def echange_lignes(A, i, j):
    nc = len(A[0])
    for k in range(nc):
        A[i][k], A[j][k] = A[j][k], A[i][k]
```

Enfin viennent les transvections. Comme pour la fonction précédente, on agira par effet de bord sur la matrice fournie en entrée, en ne renvoyant rien. En particulier, l'appel se fera via `transvection_ligne(A,i,j,mu)`, et non via `A = transvection_ligne(A,i,j,mu)` :

```
def transvection_ligne(A, i, j, mu):
    """ L_i <- L_i + mu.L_j """
    nc = len(A[0]) # le nombre de colonnes
    for k in range(nc):
        A[i][k] += mu * A[j][k]
```

Exercice 7.8 Pourquoi μ et pas λ ? :-)

7.2.2 Recoller les morceaux

Grâce à tous ces outils, l'écriture du programme Python devient comme prévu un simple exercice de traduction. On commence par faire une copie de la matrice fournie en entrée [7].

ATTENTION Pourquoi faire une copie ?

La fonction de résolution de système doit être accompagnée d'une notice claire. S'il semble évident que cette fonction prendra en entrée les données du système et renverra la solution, le statut des données doit être précisé. Ici, on choisit de ne pas modifier les données fournies. En particulier, celui qui appelle la résolution est assuré que les matrices fournies ne seront pas modifiées. Le programme de résolution commence donc par faire une copie desdites matrices, pour pouvoir manipuler les lignes.

```
def resolution(A0, Y0):
    """Résolution de A0.X=Y0; A0 doit etre inversible"""
    A, Y = copie_matrice(A0), copie_matrice(Y0)
    n = len(A)
    assert len(A[0]) == n
    # Mise sous forme triangulaire
    for i in range(n):
        j = chercher_pivot(A, i)
        if j > i:
            echange_lignes(A, i, j)
            echange_lignes(Y, i, j)
        for k in range(i+1, n):
            x = A[k][i] / float(A[i][i])
            transvection_ligne(A, k, i, -x)
            transvection_ligne(Y, k, i, -x)
```

➡

7. La fonction **copie_matrice** a été écrite dans le chapitre précédent; on peut la remplacer par la fonction **deepcopy** de la bibliothèque **copy**.

```
    # Phase de remontée
    X = [0.] * n
    for i in range(n-1, -1, -1):
        X[i] = (Y[i][0]-sum(A[i][j]*X[j] for j in range(i+1,n))) / A[i][i]
    return X
```

EN PRATIQUE Dans les versions Python 2.x

La conversion de $a_{i,i}$ en flottant est nécessaire si la matrice en entrée est constituée d'entiers, car en Python 2.x, la division est vue comme une division entière : c'est le quotient dans la division euclidienne qui est renvoyé et non le quotient flottant.

Le choix qui a été fait dans Python 3.x est le suivant : l'opérateur / voit ses paramètres comme des flottants et renvoie un flottant. Ce choix de conception est pragmatique dans ce contexte, mais il fait perdre un peu en généricité. Par exemple, une résolution manipulant des rationnels « exacts » devrait être écrite sans cette conversion.

Il est à noter enfin que la phase de remontée a été accélérée grâce à l'utilisation de la fonction `sum` de Python. Le programme complet est donné programme 4 ci-après.

On peut vérifier que tout se passe bien, avec l'exemple 1 :

```
In [4]: resolution([[2,2,-3],[-2,-1,-3],[6,4,4]],[[2],[-5],[16]])
Out[4]: [-14.000000000000036, 21.000000000000046, 4.000000000000007]
```

Si on tente de résoudre les systèmes suivants, qui ne sont pas de Cramer, on a bien entendu quelques ennuis :

$$\begin{cases} x+2y &= 1 \\ 2x+4y &= 1 \end{cases} \qquad \begin{cases} x+2y &= 1 \\ 2x+4y &= 2 \end{cases}$$

```
In [5]: resolution([[1,2],[2,4]],[[1],[1]])
ZeroDivisionError: float division by zero
```

```
In [6]: resolution([[1,2],[2,4]],[[1],[2]])
ZeroDivisionError: float division by zero
```

POUR ALLER PLUS LOIN Une erreur peu parlante

Un logiciel de calcul formel traiterait ces deux situations de façon différente : dans le premier cas, l'ensemble vide serait renvoyé, alors que dans le second serait renvoyé un ensemble paramétré sous une forme ou une autre.

PROGRAMME 4 Pivot de Gauss

```
def chercher_pivot(A, i):
    n = len(A) # le nombre de lignes
    j = i # la ligne du maximum provisoire
    for k in range(i+1, n):
        if abs(A[k][i]) > abs(A[j][i]):
            j = k # un nouveau maximum provisoire
    return j

def transvection_ligne(A, i, j, mu):
    """ L_i <- L_i + mu.L_j """
    nc = len(A[0]) # le nombre de colonnes
    for k in range(nc):
        A[i][k] += mu * A[j][k]

def echange_lignes(A, i, j):
    nc = len(A[0])
    for k in range(nc):
        A[i][k], A[j][k] = A[j][k], A[i][k]

def resolution(A0, Y0):
    """Résolution de A0.X=Y0; A0 doit etre inversible"""
    A, Y = copie_matrice(A0), copie_matrice(Y0)
    n = len(A)
    assert len(A[0]) == n
    # Mise sous forme triangulaire
    for i in range(n):
        j = chercher_pivot(A, i)
        if j > i:
            echange_lignes(A, i, j)
            echange_lignes(Y, i, j)
        for k in range(i+1, n):
            x = A[k][i] / float(A[i][i])
            transvection_ligne(A, k, i, -x)
            transvection_ligne(Y, k, i, -x)
    # Phase de remontée
    X = [0.] * n
    for i in range(n-1, -1, -1):
        X[i] = (Y[i][0]-sum(A[i][j]*X[j] for j in range(i+1,n))) / A[i][i]
    return X
```

7.2.3 Comparaison avec numpy

La bibliothèque **numpy** est très performante pour le calcul scientifique. Elle réalise des calculs optimisés sur des tableaux (**array**).

En voici une utilisation élémentaire, qui reprend la résolution de l'exemple 1 :

```
In [7]: numpy.linalg.solve([[2,2,-3],[-2,-1,-3],[6,4,4]],[[2],[-5],[16]])

Out[7]: array([[-14.], [ 21.], [ 4.]])
```

On note que le résultat renvoyé est un **array**, mais que **numpy** accepte en entrée des listes de listes [8]. La précision peut sembler meilleure qu'avec le programme précédent, mais c'est en fait un leurre d'affichage :

```
In [8]: numpy.linalg.solve([[2,2,-3],[-2,-1,-3],[6,4,4]],[[2],[-5],[16]]) [0][0]

Out[8]: -14.000000000000023
```

D'une manière générale, on peut faire le pari que les résolutions effectuées par **numpy** seront plus performantes que celles que l'on va coder soi-même sans finesse, tant pour la qualité du résultat que pour le temps de calcul.

Pour autant, on évitera de s'y fier comme à une vérité indiscutable. Un *crash-test* classique [9] en calcul numérique matriciel consiste à inverser les matrices de Hilbert. On verra que ces matrices sont très mal conditionnées et que tout calcul numérique portant sur de telles matrices peut induire de grosses erreurs d'approximation.

La matrice de Hilbert d'ordre $n \in \mathbb{N}^*$ est la matrice $H_n \in \mathcal{M}_n(\mathbb{R})$ de terme général $h_{i,j} = \frac{1}{i+j-1}$ pour $1 \leqslant i, j \leqslant n$. On peut la définir par exemple ainsi (en pensant au décalage d'indice) :

```
def hilbert(n):
    return [[1./(i+j+1) for j in range(n)] for i in range(n)]
```

En notant C la matrice de $\mathcal{M}_{n,1}(\mathbb{R})$ avec des zéros partout, sauf un « 1 » en dernière position, la résolution de $H_n X = C$ va renvoyer théoriquement la dernière colonne de H_n^{-1}. Un résultat classique est le caractère entier de H_n^{-1}. Par exemple, la « dernière » composante de H_{10}^{-1} (calculée avec un logiciel de calcul formel) est 44 914 183 600 :

```
In [9]: Y = [[0]]*9 + [[1]]

In [10]: resolution(hilbert(10),Y)[9]

Out[10]: 44909661051.85882

In [11]: numpy.linalg.solve(hilbert(10),Y)[9][0]

Out[11]: 44908633925.999527
```

Pour $n = 20$, c'est bien pire. Le résultat attendu est 48722219250572027160000 et ceux renvoyés sont respectivement 368362187621445.56 (**resolution**) et −7669635576970050.0 (**solve**) !

8. D'une manière générale, Python est plutôt de bonne composition avec les types !
9. [Higham], chapitre 28, explique que c'est plus subtil.

Voici les temps de calcul en secondes, sur un ordinateur de puissance moyenne, de la résolution de $H_n X = C$, pour différentes valeurs de n :

n	50	100	200	400	800
`resolution`	0.028	0.17	1.31	10.3	82.9
`numpy.linalg.solve`	0.0012	0.0034	0.014	0.065	0.37

On reviendra dans le paragraphe suivant sur l'évolution de ces temps de calcul en fonction de n mais on peut déjà constater que `numpy` est très rapide. En réalité, les fonctions de cette bibliothèque ne sont pas écrites en Python mais font plutôt appel à d'autres bibliothèques très optimisées écrites en C ou Fortran.

7.3 Complexité

On s'intéresse ici à la *complexité* de l'algorithme du pivot de Gauss, c'est-à-dire au nombre d'opérations élémentaires effectuées lors de la résolution d'un système à n équations et n inconnues. Cette complexité dépend bien entendu de n ; mais comment ?

7.3.1 Mise sous forme triangulaire

Il s'agit d'analyser la première phase de l'algorithme, qu'on rappelle ici :

pour *i de* 0 *à* $n-2$ **faire**
  Trouver j entre i et $n-1$ tel que $|a_{j,i}|$ soit maximale.
  Échanger L_i et L_j (coefficients de la matrice et membres de droite).
  pour *k de* $i+1$ *à* $n-1$ **faire**
    $L_k \leftarrow L_k - \dfrac{a_{k,i}}{a_{i,i}} Li$

On va dans un premier temps évaluer cette complexité de façon assez fine et on donnera ensuite une version « allégée », largement suffisante en première approximation [10].

Pour chaque valeur de $i \in [\![0, n-2]\!]$:

- La recherche de j coûte $n-i$ comparaisons.
- L'échange éventuel des deux lignes coûte $2n+2$ affectations.
- Pour chaque valeur de k entre $i+1$ et $n-1$, la transvection coûte $2n+2$ affectations et autant de divisions, multiplications et soustractions.

10. Et en fait bien au-delà.

Si on décide de désigner par *opération élémentaire* chaque comparaison, affectation ou opération arithmétique sur les flottants, le coût à i fixé est majoré par $(n-i)+(2n+2)+(n-1-i)(2n+2)$, c'est-à-dire $(2n+3)(n-i)$. En sommant sur les $i \in [\![0, n-1]\!]$, on trouve donc un coût majoré par $(2n+3)\dfrac{n(n+1)}{2} = n^3 + \dfrac{5}{2}n^2 + \dfrac{3}{2}n$. Comme d'habitude on ne s'intéresse qu'au terme dominant : la complexité dans le pire des cas est équivalente à n^3.

Pour être plus rapide sans réellement perdre en pertinence, on peut se contenter de dire : *« Pour chaque i, la recherche du pivot puis l'échange de ligne ont une complexité linéaire en n, ainsi que chacune des transvections (il y en a au plus n), d'où une complexité quadratique à i fixé. Puisque i décrit* $[\![0, n-2]\!]$, *on a finalement une complexité en* n^3. »

7.3.2 Phase de remontée

Voici la deuxième et dernière phase :

pour i *de* $n-1$ *à* 0 **faire**
  pour k *de* $i+1$ *à* $n-1$ **faire**
    $y_i \leftarrow y_i - a_{i,k}x_k$
  $x_i \leftarrow \dfrac{y_i}{a_{i,i}}$

Cette fois, on a un coût clairement quadratique : à i fixé, on réalise de l'ordre de n opérations élémentaires (la fonction `sum` de Python n'est pas magique : son temps d'exécution est linéaire en la taille de la liste à sommer).

Finalement, le coût total d'une résolution est de l'ordre de n^3. Il est à noter que ce coût est principalement payé lors de la mise sous forme triangulaire.

On reprend maintenant les résultats chronométrés de la section 7.2.3. Lorsque n est multiplié par 2 :

- Le temps de calcul de la fonction `resolution` est multiplié à peu près par 8 : il s'agit bien d'une complexité cubique.
- Celui de `solve` est multiplié par quelque chose de beaucoup plus fluctuant, mais sensiblement inférieur à 8. La documentation en ligne de Python assure que cette résolution est déléguée à la bibliothèque Fortran LAPACK via une décomposition LU. Une telle décomposition est a priori cubique... On atteint les limites de l'analyse au chronomètre, peu probante ici.

7.3.3 Peut-on faire mieux que n^3 ?

Quand on pratique par exemple la méthode des éléments finis (pour résoudre des équations aux dérivées partielles), on peut être confronté à des systèmes à n équations et n inconnues

pour lesquels n est très grand, par exemple de l'ordre du milliard. Une résolution « en n^3 » est alors absolument exclue.

En pratique Temps de calcul et complexité

On peut retenir comme ordre de grandeur qu'un ordinateur personnel va réaliser de l'ordre de 10^9 opérations élémentaires dans une minute. Par exemple, un algorithme en $O(n^2)$ s'exécutera en temps raisonnable si $n = 10^4$, mais est à proscrire si $n = 10^7$.

Pour une application industrielle dont le calcul peut durer de l'ordre de quelques jours dans un centre de calcul dédié [11], on peut imaginer un nombre d'opérations élémentaires allant jusqu'à 10^{18} voire 10^{20}, mais certainement pas 10^{25}. Par ailleurs, si on doit traiter un système à n équations et n inconnues en manipulant sa matrice à n^2 entrées, la question de la gestion de la mémoire va devenir problématique si n est par exemple de l'ordre de 10^6. Ces systèmes ne sont donc en général pas traités avec une représentation « pleine » des matrices associées.

On dispose de deux types d'améliorations :

- Des méthodes itératives permettent d'*approcher* les solutions. Dans le cadre fréquent des matrices creuses (de taille $n \times n$, mais avec un nombre d'entrées non nulles de l'ordre de $K.n$), ces algorithmes ont en général un coût de l'ordre de αn^2 voire moins, avec α dépendant de la qualité de l'approximation souhaitée et des caractéristiques de la matrice. Dans ces méthodes, on peut en plus paralléliser les calculs. C'est un avantage intéressant.
- On peut tenter d'améliorer la complexité de l'inversion matricielle « exacte ».

Le premier point sera évoqué dans les exercices. Pour le second, un résultat est assez remarquable pour être signalé et plutôt surprenant :

Théorème. Si on note $M(n)$ la complexité (dans le pire des cas) du calcul du produit de deux matrices (n, n), et sous l'hypothèse raisonnable $n^2 = O(M(n))$, alors on sait inverser toute matrice (n, n) en temps $O(M(n))$. *Bref : inverser ne coûte pas plus cher que multiplier.*

Démonstration. La première étape, simple, consiste à se ramener au cas des matrices symétriques définies positives via l'écriture $A^{-1} = \left({}^tA.A\right)^{-1}\,{}^tA$. Ensuite, lorsque B est symétrique définie positive, on l'écrit par blocs de tailles $(n/2, n/2)$ puis on effectue le calcul de B^{-1} en inversant récursivement différents blocs et leur *complément de Schur.* On obtient ainsi un très joli algorithme « diviser pour régner »... dont le lecteur curieux trouvera les détails par exemple dans [Cormen]. □

La multiplication *naïve* requiert de l'ordre de n^3 opérations élémentaires. Une amélioration classique utilisant le principe « diviser pour régner » permet de descendre cette complexité

11. Les très gros ordinateurs actuels réalisent de l'ordre de 10^{15} opérations sur les flottants par seconde ; on parle de « petaflops ».

à n^α avec $\alpha = \ln_2(7) \simeq 2,69$ (algorithme de Strassen, 1972). Depuis, on a fait un peu mieux, mais les constantes multiplicatives sont telles qu'elles rendent ces méthodes peu utilisables. La question « Peut-on, pour tout $\alpha > 2$, trouver un algorithme de multiplication de complexité $O(n^\alpha)$? » reste en particulier ouverte.

Enfin, on peut noter que certaines matrices avec une *géométrie* particulière donnent lieu à des résolutions de systèmes simplifiées. Par exemple, la matrice tridiagonale [12] suivante :

$$V_n = \begin{pmatrix} 2 & -1 & & & (0) \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ (0) & & & -1 & 2 \end{pmatrix}$$

demande une seule transvection pour chaque pivot, d'où un calcul d'inverse (ou de résolution de $V_n X = Y$) de complexité quadratique. Cette « matrice de Virginie [13] » intervient dans les schémas numériques de résolution d'équations différentielles telles que $\Delta f = g$. Elle est très classique en analyse numérique. Des variantes « tridiagonales par blocs » existent, en particulier pour résoudre les équations aux dérivées partielles en dimension 2 ou 3.

SAVOIR-FAIRE Tenir compte des aspects pratiques

Un algorithme est la plupart du temps une version idéalisée d'une procédure de résolution d'un problème. Lorsque l'on s'attache à le traduire sous forme d'un programme, on doit prendre en compte différentes considérations pratiques externes à l'algorithme proprement dit, notamment :

- les conséquences des erreurs d'arrondi sur les résultats,
- le temps de calcul,
- le stockage des données en mémoire.

La première affecte la confiance que l'on peut avoir dans les résultats fournis par un programme ; si les deux dernières sont trop critiques, l'algorithme n'a qu'un intérêt théorique.

On l'a vu sur l'exemple du pivot de Gauss, les méthodes numériques peuvent rapidement poser problème de ces trois points de vue.

12. Qu'on rencontrera régulièrement dans la suite de ce chapitre.
13. Dénomination classique, bien que d'origine peu claire.

7.4 Conditionnement d'une matrice

Les aspects théoriques évoqués dans cette section ne sont pas formellement au programme et nécessitent des connaissances de deuxième année.

Exercice 7.9 Résoudre les systèmes $AX = Y$, avec $A = H_5$ (matrice de Hilbert, voir la section 7.2.3) et $Y = \begin{pmatrix} -7.7 \\ -6 \\ -2.1 \\ -0.5 \\ 0 \end{pmatrix}$, puis $Y = \begin{pmatrix} -7.7 \\ -6 \\ -2.1 \\ -0.4 \\ 0 \end{pmatrix}$. Que doit-on en conclure ?

7.4.1 Mesurer les propagations d'erreurs

Comme cela a déjà été évoqué, le résultat d'une résolution de $AX = Y$ comporte en général des erreurs de trois types, même en suivant un algorithme à la lettre : celles sur la donnée Y, celles sur la matrice A et enfin celles qui vont apparaître en cours d'exécution, liées aux calculs en arithmétique flottante. À défaut de pouvoir supprimer ces erreurs, on aimerait avoir un certain contrôle sur leur propagation.

Il est assez délicat de mesurer l'ensemble de ces erreurs, ce qui conduit à quelques raccourcis.

Même en imaginant des données connues exactement, ce qui est une hypothèse très optimiste, les premières opérations arithmétiques induiront possiblement une erreur relative de l'ordre de 2^{-k}, avec k la taille de la mantisse. Ceci est vrai pour A comme pour Y.

On cherche donc, ensuite, à contrôler l'erreur (relative) sur X à l'aide des erreurs relatives sur A et Y.

Pour cela, on munit $\mathcal{M}_{n,1}(\mathbb{R})$ (assimilé à $\mathbb{R}^n$) d'une norme (par exemple la norme euclidienne canonique) et $\mathcal{M}_n(\mathbb{R})$ de la norme subordonnée. On suppose que $A_0X_0 = Y_0$ (le problème initial) et $A_1X_1 = Y_1$ (le système qu'on a résolu), avec $\delta A = A_1 - A_0$ et $\delta Y = Y_1 - Y_0$ respectivement petits devant A_0 et Y_0. On souhaite, en notant $\delta X = X_1 - X_0$, majorer $\dfrac{\|\delta X\|}{\|X_0\|}$ à l'aide de $\dfrac{|||\delta A|||}{|||A_0|||}$ et $\dfrac{\|\delta Y\|}{\|Y_0\|}$.

7.4.2 Le conditionnement

Observons ce qui se passe dans un cas très simple.

Exercice 7.10 Conditionnement : cas diagonal, erreur sur le membre de droite.
On suppose que A est une matrice inversible diagonale, $A = Diag(\lambda_1, ..., \lambda_n)$, avec de plus :

$$0 < |\lambda_1| \leqslant \cdots \leqslant |\lambda_n| .$$

On suppose : $AX = Y$, $A(X + \delta X) = Y + \delta Y$ et on munit $\mathbb{R}^n$ de la norme euclidienne canonique.

Montrer :

$$\frac{\|\delta X\|}{\|X\|} \leqslant \frac{|\lambda_n|}{|\lambda_1|} \times \frac{\|\delta Y\|}{\|Y\|}.$$

Exercice 7.11 Conditionnement : cas diagonal, erreur sur la matrice. On suppose que A et δA sont des matrices diagonales avec les notations $A = Diag(\lambda_1, ..., \lambda_n)$ et $\delta A = Diag(\delta\lambda_1, ..., \delta\lambda_n)$, et de plus :

$$0 < |\lambda_1| \leqslant \cdots \leqslant |\lambda_n|.$$

On suppose : $AX = Y$, $(A + \delta A)(X + \delta X) = Y$ et on munit $\mathbb{R}^n$ de la norme euclidienne canonique.
Montrer :

$$\frac{\|\delta X\|}{\|X + \delta X\|} \leqslant \frac{|\lambda_n|}{|\lambda_1|} \times \frac{|||\delta A|||}{|||A|||},$$

avec $|||\ |||$ la norme subordonnée à $\|\ \|$.

Dans les exercices précédents, λ_n est la valeur propre la plus grande (en module), mais son module est aussi la norme de A (pour la norme subordonnée à la norme euclidienne sur $\mathbb{R}^n$). De même, $\dfrac{1}{|\lambda_1|}$ est égal à $|||A^{-1}|||$. Ce sont bien ces quantités qui, dans le cas général, vont permettre de quantifier les propagations d'erreurs dans les résolutions de système.

Si $A \in \mathcal{M}_n(\mathbb{R})$ est inversible, son *conditionnement*, noté en général $\kappa(A)$, associé à une norme $\|\ \|$ de $\mathbb{R}^n$ est défini par :

$$\kappa(A) = |||A|||.|||A^{-1}|||.$$

Avec cette définition, on retrouve dans le cas général les résultats vus plus haut dans un cas particulier.
Théorème. Contrôle des erreurs par le conditionnement.

- Si $AX = Y$ et $A(X + \delta X) = Y + \delta Y$, alors $\dfrac{\|\delta X\|}{\|X\|} \leqslant \kappa(A)\dfrac{\|\delta Y\|}{\|Y\|}$.
- Si $AX = Y$ et $(A + \delta A)(X + \delta X) = Y$, alors $\dfrac{\|\delta X\|}{\|X + \delta X\|} \leqslant \kappa(A)\dfrac{|||\delta A|||}{|||A|||}$.

On peut même améliorer la dernière inégalité : $\dfrac{\|\delta X\|}{\|X\|} \leqslant \kappa(A)\dfrac{|||\delta A|||}{|||A|||}(1 + O(|||\delta A|||))$*, le grand O étant à prendre au sens de : lorsque* δA *tend vers* 0.

7.4.3 Quelques exemples caractéristiques

Ce petit bestiaire a pour objectif de rassurer le lecteur : en général, le conditionnement d'une matrice n'est pas mauvais. Il faut essentiellement craindre les matrices à petites valeurs propres.

Les faits suivants peuvent, pour la plupart, constituer de petits exercices de mathématiques de seconde année.

- Les matrices d'homothéties λI_n ont pour conditionnement[14] 1.
- Si A est orthogonale, alors $\kappa(A) = 1$.
- Pour une matrice diagonale inversible, le conditionnement est le rapport entre les modules maximal et minimal des éléments diagonaux.
- Pour une matrice symétrique réelle de valeurs propres (réelles, forcément) $\lambda_1, ..., \lambda_n$, le conditionnement est le rapport entre le maximum et le minimum des $|\lambda_i|$.

On va terminer avec deux cas particuliers intéressants : les matrices de Hilbert et de Virginie, qu'on a évoquées dans les sections précédentes.

```
for n in [10,100,1000]:
    spectre = numpy.linalg.eigvals(Virginie(n))
    print(numpy.max(spectre)/numpy.min(spectre))
48.3741500787
4133.6429268
406095.042659
```

Le spectre de V_n est en fait parfaitement connu il est constitué des réels de la forme $2\left(1+\cos\dfrac{k\pi}{n+1}\right)$, pour k décrivant $[\![1,n]\!]$. On en déduit sans trop de mal que $\kappa(V_n) \sim \dfrac{4n^2}{\pi^2}$ et, puisque $\dfrac{4}{\pi^2} \simeq 0.405$, les résultats expérimentaux semblent cohérents avec la théorie !

Exercice 7.12 avec corrigé À l'aide de la fonction `eigvals` de la sous-bibliothèque `numpy.linalg`, qui calcule les valeurs propres d'une matrice, déterminer le conditionnement de H_n, pour $n \in [\![2, 20]\!]$. Ces résultats vous semblent-ils fiables ?

```
for n in range(2,15):
    spectre = numpy.linalg.eigvals(Hilbert(n))
    kappa = numpy.max(spectre)/numpy.min(spectre)
    print(n,kappa,numpy.log(kappa)/n)
(2, 19.281470067903978, 1.4795722673054559)
(3, 524.05677758606339, 2.0872000108743429)
(4, 15513.738738929998, 2.412370320141686 2)
(5, 476607.25024226878, 2.6148896111914297)
(6, 14951058.641519276, 2.7533821112392776)
(7, 475367356.07704318, 2.854228349186942)
(8, 15257575750.674316, 2.9310427483982773)
(9, 493153514278.87506, 2.9915651500563918)
(10, 16025161728345.84, 3.0405181210936822)
(11, 522138774130152.75, 3.0808140472093792)
(12, 17115434512705980.0, 3.1148964212000259)
(13, 1.3195210596549663e+18, 3.2095231162351037)
__main__:4: RuntimeWarning: invalid value encountered in log
(14, -3.254915640839065e+17, nan)
```

14. Ici, et pour les exemples suivants, le conditionnement est donné pour la norme subordonnée à la norme euclidienne canonique de $\mathbb{R}^n$.

Les valeurs extrêmes du spectre des matrices de Hilbert sont relativement bien connues, mais les résultats (surtout l'équivalent de la plus petite) sont « non élémentaires » à démontrer !

Exercice 7.13 Comment l'exercice 7.9 a-t-il été conçu ?

7.5 Exercices

Exercice 7.14 * Démontrer rigoureusement à l'aide d'un invariant de boucle que la première phase de l'algorithme du pivot de Gauss conduit à un système sous forme triangulaire.

Exercice 7.15 Une façon naïve pour calculer le déterminant d'une matrice consiste à développer selon une ligne ou colonne, ce qui amène à un calcul récursif coûteux en général.

1 Évaluer la complexité d'un tel algorithme.

2 Si un ordinateur peut effectuer 10^9 opérations sur les flottants par seconde, jusqu'à quelle dimension peut-on espérer calculer un déterminant par cette méthode en un temps majoré par une journée ?
Une autre façon de calculer le déterminant consiste à pivoter, pour se ramener à une matrice diagonale.

3 Expliciter cet algorithme à l'aide de pseudo-code. Évaluer sa complexité.

4 Programmer effectivement cet algorithme en Python.

5 Le tester, exhiber des cas limites mettant en défaut le programme.

Le lecteur voulant tester sa virtuosité en manipulation de tableaux avec `numpy` *pourra programmer le calcul naïf du déterminant et vérifier empiriquement la complexité.*

Exercice 7.16 * En réalisant des opérations élémentaires sur les lignes (et/ou colonnes, mais on peut se contenter d'opérations sur les lignes), on peut passer d'une matrice inversible quelconque A à la matrice identité I_n. Ceci permet de calculer A^{-1}.

1 Préciser l'algorithme à l'aide de pseudo-code. Évaluer sa complexité.

2 Programmer effectivement cet algorithme en Python.

3 Tester ce programme sur des exemples tels que V_n et H_n (matrices de Virginie et de Hilbert). Comparer le résultat numérique avec celui obtenu avec `numpy`.

4 Vérifier empiriquement que le temps de calcul de ce programme est bien de l'ordre du cube de la dimension de la matrice à inverser.

Exercice 7.17 Le module `fractions` de Python fournit une représentation et des opérations pour manipuler des rationnels en valeur exacte. L'expression `Fraction(numerateur, denominateur)` construit la fraction correspondante, sur laquelle on peut ensuite utiliser les opérations usuelles. On consultera sa documentation pour plus de détails.

1 Adapter le programme Python du pivot de Gauss pour qu'il résolve des systèmes à coefficients rationnels de façon exacte.

2 Le tester sur des exemples tels que V_n et H_n (matrices de Virginie et de Hilbert).

3 Empiriquement, le temps de calcul de ce programme est-il toujours de l'ordre du cube de la dimension de la matrice à inverser ? Si ce n'est pas le cas, proposer une explication.

Exercice 7.18 * Pour tester une procédure « maison » calculant l'inverse d'une matrice, on va l'exécuter sur une matrice aléatoire, dont on peut raisonnablement espérer qu'elle sera inversible. On comparera (en termes de précicion et de rapidité) avec la fonction dédiée `numpy.linalg.inv`.

1 Évaluer la différence entre les résultats de votre procédure d'inversion et ceux de `numpy.linalg.inv`, sur des matrices de taille (n, n), avec $n \in \{10, 50, 100, 200\}$.

2 En utilisant le module `Image`, visualiser les matrices initiales et inverses.

Pour évaluer la différence, on commencera par choisir une norme matricielle. Pour visualiser, on renormalisera la matrice pour obtenir des coefficients entre 0 *et* 255*, ce qui permet de créer un fichier bitmap.*

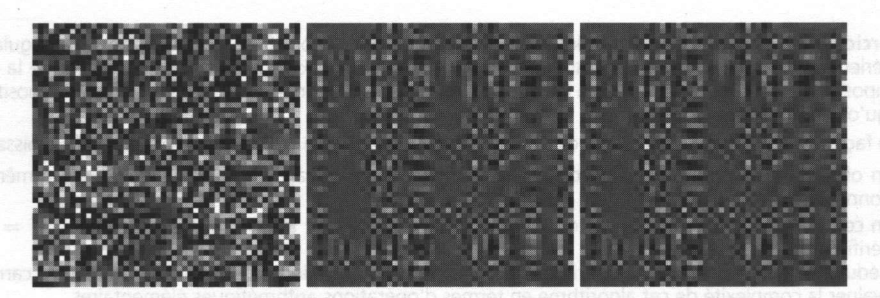


Figure 7.1
La matrice initiale, son inverse « maison » et celle calculée avec `linalg.inv`

Le lecteur intrigué par les régularités de la matrice inverse pourra aller consulter [Appel].

Exercice 7.19 * On a vu en section 7.1.5 que l'algorithme du pivot de Gauss conduit à une décomposition « lower-upper » de certaines matrices. Cet exercice précise ce point.

1. Montrer que si les n mineurs principaux d'une matrice sont non nuls, alors dans l'algorithme du pivot de Gauss (sans choix du module maximal), on trouve à chaque étape (qu'on appellera k) un élément non nul en position (k, k) dans la matrice.
2. On suppose qu'à chaque opération sur les lignes de A (la matrice qu'on veut mettre sous la forme LU), on réalise l'opération équivalente sur une matrice B initialisée à I_n.
   Montrer qu'on obtient ainsi une matrice B triangulaire inférieure telle que BA soit triangulaire supérieure.
3. On suppose que dans la question précédente $B = M_N...M_2M_1$, les M_k étant associées à des opérations sur les lignes. Que vaut alors B^{-1} ? Et comment faire en sorte de calculer cette matrice B^{-1} à la volée (pendant la mise sous forme triangulaire de A) plutôt qu'*a posteriori* ?
4. Écrire un programme Python prenant en entrée une matrice vérifiant les hypothèses faites plus haut et renvoyant deux matrices L et U, respectivement triangulaires inférieure et supérieure, telles que $A = LU$.
   On pourra comparer le résultat avec celui proposé par la fonction `lu` *du sous-module* `numpy.linalg`*.*
5. Sans l'hypothèse faite sur les mineurs de A, montrer que si A est inversible, alors on peut trouver L et U, triangulaires inférieure et supérieure, et P une matrice de permutation[15], telles que $A = LPU$ (*décomposition de Bruhat*).

Lorsque la décomposition LU est connue, la résolution cubique de $AX = Y$ se ramène à deux résolutions quadratiques de systèmes triangulaires, ce qui est très intéressant.

Attention — Encore les flottants et l'égalité

Le calcul effectif de la décomposition de Bruhat est peu pertinent en arithmétique flottante, puisqu'il est associé à l'annulation d'un coefficient de la matrice et on sait ce qu'il en est de la nullité d'un coefficient représenté par un flottant...

Il peut être intéressant tout de même d'écrire un programme réalisant la décomposition de Bruhat pour une matrice à coefficients dans $\mathbb{Q}$: en pseudo-code dans un premier temps, puis en Python en utilisant le module `fractions` dans un deuxième temps.

15. Elle ne contient que des zéros, sauf un « 1 » par ligne et par colonne ; bref, on a permuté les lignes (ou colonnes) de la matrice identité.

Exercice 7.20 * Si A est une matrice réelle symétrique définie positive, alors il existe U triangulaire supérieure telle que $A = {}^tU.U$: c'est la décomposition de *Choleski*, qui est un cas particulier de la décomposition LU. Elle est unique si on impose aux coefficients diagonaux de U d'être strictement positifs, ce qu'on va faire dans la suite.

Une façon de construire U consiste à calculer les coefficients $u_{i,j}$ à i croissant, puis (à i fixé) à j croissant.

1 En observant le coefficient $(1, 1)$ du produit ${}^tU.U$, donner la valeur nécessaire de $u_{1,1}$. De même, donner les valeurs nécessaires de $u_{1,j}$ pour $j \geqslant 2$.
2 En continuant le procédé, donner des conditions nécessaires sur tous les $u_{i,j}$ pour avoir ${}^tU.U = A$. Vérifier qu'elles sont suffisantes.
3 Déduire de ce qui précède un algorithme calculant la décomposition de Choleski d'une matrice carrée.
4 Évaluer la complexité de cet algorithme en termes d'opérations arithmétiques élémentaires.
5 Programmer en Python la décomposition de Choleski.
6 Tester, comparer avec la fonction `linalg.cholesky` de **`numpy`**.

En changeant peu de choses, cet algorithme de Choleski permet de tester le caractère positif d'une matrice symétrique... en espérant que la matrice n'ait pas de valeurs propres trop proches de 0.

POUR ALLER PLUS LOIN Décomposition QR

Toute matrice carrée[16] réelle peut se décomposer sous la forme QR, avec Q orthogonale et R triangulaire supérieure. Cette décomposition a beaucoup d'applications théoriques... mais aussi pratiques ! Par exemple, une fois la décomposition $A = QR$ connue, résoudre $AX = Y$ se décompose en deux opérations plus simples : d'abord une résolution (quadratique) de système triangulaire, puis une multiplication (elle aussi quadratique) par $Q^{-1} = {}^tQ$.

Trois points de vue conduisent à cette décomposition :

- On peut voir la matrice initiale comme représentant n vecteurs dans la base canonique de $\mathbb{R}^n$ muni de son produit scalaire euclidien canonique. Le procédé de Gram-Schmidt appliqué à cette famille de vecteurs fournit une base orthonormée. Si on s'intéresse aux matrices de passage entre les trois bases en présence, on obtient la décomposition souhaitée. Cette méthode a le bon goût d'être d'interprétation géométrique claire... mais est numériquement assez instable.
- La méthode de Householder, présentée dans l'exercice suivant, consiste à composer l'application de départ avec une réflexion (symétrie orthogonale par rapport à un hyperplan) pour que le premier vecteur soit envoyé sur un vecteur colinéaire à lui-même. On continue ensuite le travail récursivement.
- La méthode de Givens consiste, elle, à composer avec des « rotations ». Cette dernière méthode est un peu plus coûteuse, mais est plus stable que la méthode de Householder.

Exercice 7.21 * On présente ici la méthode de Householder pour trouver une décomposition QR de $A \in \mathcal{M}_n(\mathbb{R})$. Notons e_1 le premier vecteur de la base canonique de $\mathbb{R}^n$.

Si Ae_1 est colinéaire à e_1, alors on peut écrire par blocs $A = \begin{pmatrix} \alpha & L \\ (0) & A_1 \end{pmatrix}$, avec $A_1 \in \mathcal{M}_{n-1}(\mathbb{R})$. Un appel récursif fournit $Q_1 \in O_{n-1}(\mathbb{R})$ et $R_1 \in \mathcal{M}_{n-1}(\mathbb{R})$ triangulaire supérieure, telles que $A_1 = Q_1R_1$. Il suffit alors de prendre $Q = \begin{pmatrix} 1 & (0) \\ (0) & Q_1 \end{pmatrix}$ et $R = \begin{pmatrix} \alpha & L \\ (0) & Q_1 \end{pmatrix}$ pour répondre au problème.

Si Ae_1 n'est pas colinéaire à e_1, on va considérer $v = Ae_1 - \varepsilon \|Ae_1\| e_1$ (avec $\varepsilon \in \{-1, 1\}$ du signe de la première composante de Ae_1 ; condition imposée pour la stabilité). La réflexion par rapport à l'orthogonal

16. En fait, une adaptation de cette décomposition existe aussi (et est utile) pour les matrices rectangulaires quelconques.

de v va alors envoyer Ae_1 sur la droite engendrée par e_1. Cette réflexion a pour matrice dans la base canonique $S_1 = I_n - \frac{2}{\|v\|^2} v.\,{}^t v$. Ainsi, $S_1 A$ possède sa première colonne colinéaire à e_1 et on est ramené au cas précédent.

1 Formaliser l'algorithme précédent en pseudo-code.
2 Écrire un programme Python implémentant cet algorithme.
3 Quelle est la complexité de cette méthode, en termes d'opérations arithmétiques ?

Exercice 7.22 ** *On s'intéresse ici à la méthode de Jacobi, une méthode* itérative *de résolution approchée de $Ax = y$, sous des conditions assez fortes*[17] *sur A.*

Soit $A \in \mathcal{M}_n(\mathbb{R})$ une matrice dont la diagonale est dominante, c'est-à-dire : pour tout $i \in [\![1, n]\!]$, $|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$. Cette condition assure de façon classique le caractère inversible de A, donc le système $Ax = y$ est de Cramer.

On décompose A sous la forme $A = D + H$, avec D diagonale et H comportant des 0 sur la diagonale. On a alors $Ax = y$ si et seulement si $Dx = -Hx + y$, soit encore (les conditions sur A assurent que les éléments diagonaux de D sont différents de 0) : $x = -D^{-1}Hx + D^{-1}y$. La méthode de Jacobi consiste alors à choisir un premier vecteur x_0 quelconque (par exemple $x_0 = 0$), puis définir une suite $(x_p)_{p \in \mathbb{N}}$ par la relation de récurrence $x_{p+1} = -D^{-1}Hx_p + D^{-1}y$. On démontre que sous les hypothèses faites plus haut, la suite converge vers x, l'unique solution de $Ax = y$. En pratique, un point délicat consiste à déterminer une condition raisonnable d'arrêt (calculer une infinité de termes est assez lassant).

1 On admet la formule suivante, qui permet de contrôler l'erreur $\|x_p - x\|$ à l'aide de la quantité $R = \max_i \frac{1}{a_{i,i}} \sum_{j \neq i} |a_{i,j}| < 1$:

$$\|x - x_p\| \leqslant \frac{R^p}{1 - R} \|x_1 - x_0\|$$

(la norme considérée ici étant $\|z\| = \max(|z_i|)$).
Déduire de cette majoration un test d'arrêt dans la méthode de Jacobi, si on se donne pour objectif une majoration $\|x - x_p\| \leqslant \varepsilon_0$.

2 Écrire en pseudo-code cet algorithme de Jacobi.

3 Quelle est la complexité de cet algorithme (en termes d'opérations arithmétiques élémentaires et fonction de n et ε_0) ?

4 Programmer une fonction Python **`jacobi`** implémentant effectivement cet algorithme. Tester. Comparer avec la fonction **`linalg.jacobi`** de **`numpy`**.

5 La matrice de Virginie n'a pas sa diagonale dominante (les inégalités requises sont strictes[18]), mais on peut montrer que le rayon spectral[19] de $D^{-1}(V_n - D)$ est de la forme $1 - \alpha_n$, avec $\alpha_n \sim \frac{K}{n^2}$, ce qui assure une convergence avec p bits significatifs en un nombre d'itérations de l'ordre de pn^2.

Quel est le coût d'une résolution de $V_n X = Y$ à pn^2 itérations ? On distinguera selon le mode de représentation choisi : avec une matrice « pleine » (classique), ou en exploitant le caractère creux de V_n. Programmer et expérimenter la méthode de Jacobi sur ce type d'exemple peut faire l'objet de TP très riches.

17. Il existe une condition nécessaire et suffisante relativement précise qui assure la convergence, mais on a choisi de présenter ici une condition suffisante simple.

18. Elle est tout de même faiblement dominante, avec des inégalités larges, au moins une stricte et le caractère irréductible.

19. Ici : la plus grande valeur propre.

POUR ALLER PLUS LOIN Méthode de Gauss-Seidel

La méthode de Jacobi est la plus élémentaire des méthodes itératives. Ces dernières, de façon générique, partent d'une décomposition $A = D + M$, avec D « facilement » inversible[20], et on itère l'opération $x \leftarrow -D^{-1}Mx + D^{-1}y$ dans le but de s'approcher d'un point fixe de l'application $Z \mapsto -D^{-1}MZ + D^{-1}y$.

Pour la méthode de Jacobi, D est constituée uniquement des éléments diagonaux (avec des zéros en dehors de la diagonale). Une variante est la méthode de Gauss-Seidel : il s'agit simplement de prendre pour D la matrice constituée des éléments diagonaux et sous-diagonaux de A. En pratique, cela revient à calculer la i-ème composante de $x^{(k)}$ en utilisant les composantes $x_j^{(k+1)}$ déjà calculées pour $j < i$ et les $x_j^{(k)}$ pour $j \geqslant i$. Cette simple modification améliore la convergence.

La méthode de Gauss-Seidel est beaucoup plus délicate à paralléliser, mais ça reste possible en *pipelinant* les calculs. C'est très technique, mais le jeu en vaut la chandelle !

Exercice 7.23 * On s'intéresse ici au calcul du polynôme minimal d'une matrice carrée $A \in \mathcal{M}_n(\mathbb{K})$, c'est-à-dire le polynôme unitaire P de plus petit degré tel que $P(A) = 0$. On sait qu'un tel polynôme existe et est de degré majoré par n.

1 Un algorithme simple pour calculer le polynôme minimal d'une matrice $A \in \mathcal{M}_n(\mathbb{K})$ consiste à calculer $A^2, \dots, A^n$, placer leurs coefficients (ainsi que ceux de I_n) dans une matrice $B \in \mathcal{M}_{n^2,n}(\mathbb{K})$, pivoter sur les colonnes jusqu'à obtenir une colonne nulle, fournissant ainsi une combinaison linéaire nulle non triviale et minimale des A^k.

a) Détailler, en écrivant le pseudo-code de l'algorithme.

b) Montrer qu'on obtient ainsi un calcul du polynôme minimal en $O(n^4)$ opérations élémentaires.

c) Expliquer pourquoi, numériquement, cet algorithme est voué à l'échec.[21]

2 (Difficile) Proposer un algorithme probabiliste permettant d'obtenir en $O(n^3)$ opérations arithmétiques le polynôme minimal d'une matrice « avec une forte probabilité »... en un sens à préciser !
On pourra tirer un vecteur au hasard et calculer le polynôme minimal de A vis-à-vis de ce vecteur.

Exercice 7.24 **** *Pour rire.*

Le *Permanent* d'une matrice $A \in \mathcal{M}_n(\mathbb{K})$ est défini par : $\sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$

Trouver un algorithme calculant le permanent d'une matrice de $\mathcal{M}_n(\mathbb{K})$ en temps polynomial en n.

ATTENTION Un exercice impossible ?

On évitera de passer trop de temps sur l'exercice précédent... Le lecteur intéressé pourra faire une recherche sur le mot-clé « ♯P-complet » (prononcer « sharp P-complet »). Quelqu'un sachant calculer le permanent en temps polynômial sait répondre rapidement à la question suivante : « combien de n-uplets de booléens $(b_1, \dots, b_n)$ satisfont telle formule booléenne (de longueur $O(n)$) ? ». Cette personne sait *a fortiori* répondre rapidement à la question « Telle formule booléenne de longueur $O(n)$ est-elle satisfaisable ? ». Fortune et gloire sont promises à cette personne.

20. Au sens : la résolution de $DX = Y$ est simple.

21. Si on fait du calcul formel dans $\mathcal{M}_n(\mathbb{Q})$ par exemple, il est tout à fait valable. Encore une idée de TP !

8

Résolution numérique d'équations sur les réels

Nous allons voir dans ce chapitre comment évaluer numériquement une solution d'équation de la forme $f(x) = 0$ avec f une application de $\mathbb{R}$ dans $\mathbb{R}$, dans des cadres où la théorie nous assure qu'une telle solution existe, mais sans en fournir d'expression analytique.

Deux méthodes vont être présentées :

- La méthode de dichotomie approche une solution avec p bits significatifs en p étapes, ce qui est déjà très efficace. Ses conditions d'utilisation sont assez simples : on demande seulement à la fonction d'être continue et de changer de signe. Ceci en fait une méthode robuste.
- La méthode de Newton a une vitesse de convergence assez diabolique : à chaque étape supplémentaire, le nombre de bits (ou décimales) significatifs correct(e)s est multiplié par deux ! On atteint par exemple une précision de 50 bits en sept étapes. En contrepartie, elle nécessite un ensemble de conditions d'application parfois délicat à vérifier.

Une bonne compréhension de ces deux méthodes aidera à faire un choix raisonné. Comme dans le chapitre précédent, on programmera et testera ces algorithmes, puis on les comparera aux performances des fonctions fournies par la bibliothèque **numpy**.

8.1 Méthode dichotomique

8.1.1 Principe théorique

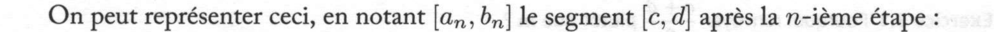
Lorsque f est une fonction continue sur un intervalle $[a, b]$, à valeurs réelles, avec $f(a)$ et $f(b)$ de signe (large) opposé, le théorème des valeurs intermédiaires nous assure que f s'annule entre a et b. Une démonstration élégante consiste à considérer la borne supérieure de l'ensemble des $x \in [a, b]$ tels que $f(x)$ soit du signe de $f(a)$: il est assez simple de montrer que f s'annule en ce point.

Si une telle démonstration est limpide à rédiger, la démonstration dichotomique a le bon goût d'être *constructive*, au sens où elle fournit un algorithme pour approcher une solution. Il s'agit de construire par récurrence une suite de segments emboîtés, qui va « converger » vers un singleton $\{l\}$, et on démontre alors que $f(l) = 0$. Le point crucial est de préserver la propriété (ou *invariant*, si on pense en informaticien) :

f change de signe entre les deux extrémités du segment.

Exemple 1. Le cas de $f : x \mapsto x^2 - 2$ est simple et démonstratif.

- $f(1) = -1 < 0 < 2 = f(2)$, donc l'équation $f(x) = 0$ possède une solution dans $[1, 2]$. La valeur médiane de cet intervalle est 1.5.
- $f(1.5) = 0.25 > 0 > f(1)$, donc l'équation $f(x) = 0$ possède une solution dans $[1, 1.5]$.
- $f(1.25) = -0.4675 < 0 < f(1.5)$, donc l'équation $f(x) = 0$ possède une solution dans $[1.25, 1.5]$.
- ...
- $f(1.41430664062) \simeq 0.000263273715973 > 0 > f(1.4140625)$, donc l'équation $f(x) = 0$ possède une solution dans $[1.4140625, 1.41430664062]$.



On peut représenter ceci, en notant $[a_n, b_n]$ le segment $[c, d]$ après la n-ième étape :

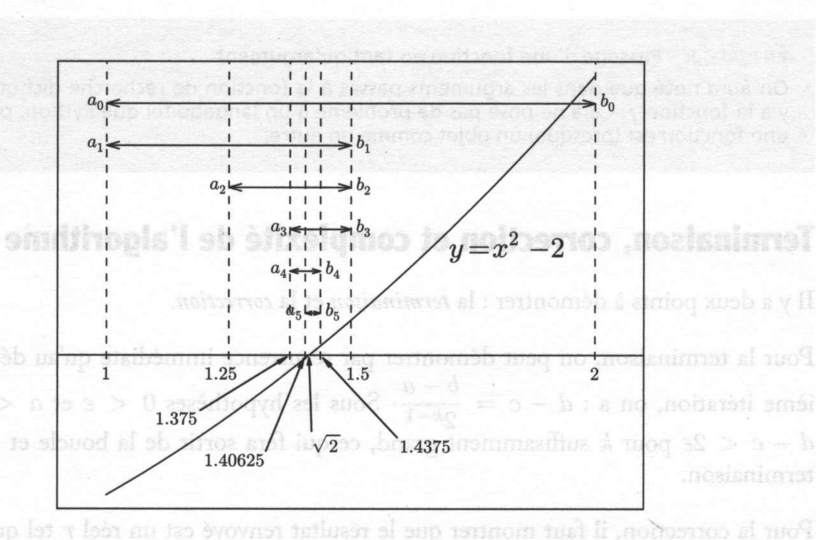


Figure 8.1
Les cinq premières itérations du calcul approché de $\sqrt{2}$ par dichotomie

On arrête le processus lorsque $d - c$ a atteint une valeur correspondant à la précision demandée (ou bien son double, si on décide de renvoyer $\dfrac{c+d}{2}$).

On va ainsi mettre en place l'algorithme suivant :

Données : f, a, b, ε
$c, d \leftarrow a, b$
$f_de_c, f_de_d \leftarrow f(c), f(d)$
tant que $d - c > 2\varepsilon$ **faire**
  $m \leftarrow (c+d)/2$
  $f_de_m \leftarrow f(m)$
  si $f_de_c \times f_de_m \leqslant 0$ **alors**
    $d \leftarrow m$
    $f_de_d \leftarrow f_de_m$
  sinon
    $c \leftarrow m$
    $f_de_c \leftarrow f_de_m$

Résultat : $\dfrac{c+d}{2}$

Exercice 8.1 Pourquoi renvoyer $\frac{c+d}{2}$ plutôt que m ?

> En pratique **Passage d'une fonction en tant qu'argument**
> On aura noté que dans les arguments passés à la fonction de recherche dichotomique, il y a la fonction f. Cela ne pose pas de problème à un langage tel que Python, pour lequel une fonction est (presque) un objet comme un autre.

8.1.2 Terminaison, correction et complexité de l'algorithme

Il y a deux points à démontrer : la *terminaison* et la *correction.*

Pour la terminaison, on peut démontrer par récurrence immédiate qu'au début de la k-ième itération, on a : $d - c = \frac{b-a}{2^{k-1}}$. Sous les hypothèses $0 < \varepsilon$ et $a < b$, on aura $d - c < 2\varepsilon$ pour k suffisamment grand, ce qui fera sortir de la boucle et démontre la terminaison.

Pour la correction, il faut montrer que le résultat renvoyé est un réel r tel que l'équation $f(x) = 0$ possède une solution x_0 telle que $|x_0 - r| \leqslant \varepsilon$. La clé de la démonstration est l'*invariant* suivant :

À chaque itération, on a $f(c)f(d) \leqslant 0$.

Pour démontrer cet invariant, on note qu'il est bien vérifié par hypothèse avant l'entrée dans la boucle [1]. Ensuite (induction), on suppose la propriété vraie au début d'une itération. Si $f(c)f(m) \leqslant 0$, alors on change la valeur de d en m sans toucher à c, donc on a bien $f(c)f(d) \leqslant 0$ à la fin de l'itération (donc au début de la suivante). Si $f(c)f(m) > 0$, la valeur de d est inchangée et on remplace c par m. Puisque $f(m)$ a le même signe (strict) que $f(c)$, le signe de $f(c)f(d)$ ne change pas dans l'affectation $c \leftarrow m$ et, comme cette quantité était négative au début de l'itération, elle l'est encore à la fin de celle-ci.

Ainsi, le réel r renvoyé est le milieu d'un intervalle de longueur majorée par 2ε (condition de sortie de boucle) et qui contient un zéro x_0 de f (grâce à l'invariant de boucle et au théorème des valeurs intermédiaires : la fonction en jeu était supposée continue). On a alors bien $|x_0 - r| \leqslant \varepsilon$.

Enfin, on va s'intéresser à la *complexité*. Ici, il convient de préciser ce qu'on prend en compte : comptabiliser les opérations arithmétiques usuelles de la même façon que les appels de f serait imprudent (chaque appel de f peut être non élémentaire). Les opérations arithmétiques élémentaires comme les appels de la fonction sont contrôlés par le nombre de passages dans la boucle.

1. Si cette hypothèse cruciale n'est pas vérifiée, le comportement de l'algorithme n'est pas spécifié. On peut arranger un peu cela grâce à une vérification de propriété en début de programme (**assert** dans ce qui suit).

La démonstration de la terminaison donne facilement cette complexité. En effet, il y a une k-ième itération si et seulement si $\frac{b-a}{2^{k-1}} > 2\varepsilon$, c'est-à-dire $k < \ln_2\left(\frac{b-a}{\varepsilon}\right)$. La boucle sera donc exécutée exactement $\left\lceil \ln_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil - 1$ fois.

Pour avoir p bits significatifs, si la solution recherchée est de l'ordre de 1, on prend $\varepsilon = \frac{1}{2^p}$ et le nombre d'itérations requises est alors de l'ordre de p comme annoncé dans le préambule.

8.1.3 Mise en place, essais

C'est à nouveau un simple exercice de traduction. Il faut noter le **assert**, qui va provoquer une erreur particulière *AssertionError* si la condition n'est pas vérifiée à l'exécution.

Programme 5 Recherche approchée d'une solution d'une équation par dichotomie

```
def dichotomie(f, a, b, epsilon):
    assert f(a) * f(b) <= 0 and epsilon > 0
    c, d = a, b
    fc, fd = f(c), f(d)
    while d - c > 2 * epsilon:
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0:
            d, fd = m, fm
        else:
            c, fc = m, fm
    return (c + d) / 2.
```

Reprenons l'exemple 1. On pourrait définir une fonction `def g(x): return x**2-2`, mais c'est inutile : Python offre la possibilité de définir des *fonctions anonymes* via l'opérateur `lambda` :

```
In [1]: math.sqrt(2), dichotomie(lambda x : x**2-2, 1, 2, 0.000001)
Out[1]: (1.4142135623730951, 1.4142141342163086)
```

Pour le deuxième exemple, on utilise la fonction `sin` de la bibliothèque `math` qu'on aura préalablement importée :

```
In [2]: math.pi, dichotomie(math.sin, 3, 4, 10**-10)
Out[2]: (3.141592653589793, 3.141592653642874)
```

Visualisons enfin les conséquences du choix de la valeur renvoyée. Le programme qu'on a écrit plus haut assure d'avoir $|x_0 - r| \leqslant \varepsilon$. Cependant, avec ce choix, la précision peut devenir moins bonne quand la valeur de ε diminue. Si on fait le choix de renvoyer celui

de c et d dont l'image est la plus faible, on fait un appel de plus à `f`, mais on obtient de façon presque certaine un résultat dont la précision s'améliorera lorsque ε diminuera (voir figure 8.2).

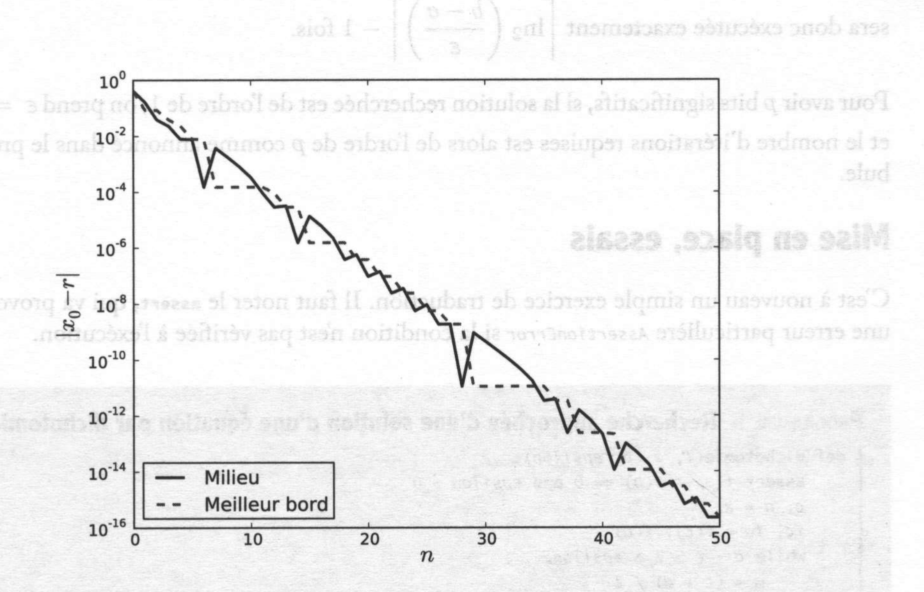


Figure 8.2
Précision, en fonction du choix de la valeur renvoyée et du nombre d'itérations

Exercice 8.2 Remplacer la valeur renvoyée par une moyenne de c et d pondérée à l'aide de $f(c)$ et $f(d)$ (pour privilégier celui dont l'image est la plus faible). Comparer avec les deux autres possibilités discutées.

8.2 Méthode de Newton

8.2.1 Extraction de racine

On reprend l'exemple 1 en partant d'une valeur initiale $u_0 = 2$ et on suit la tangente du graphe de $f : x \mapsto x^2 - 2$ depuis le point d'abscisse 2. L'intersection de cette tangente avec l'axe des abscisses donne une nouvelle valeur $u_1 = 1.5$ — qui aurait été un zéro de f si cette dernière avait été affine. On reprend la tangente depuis le point d'abscisse u_1, pour trouver une nouvelle valeur $u_2 \simeq 1.4167$. Ce procédé est illustré figure 8.3.

On se convainc facilement que la suite $(u_n)_{n\in\mathbb{N}}$ va converger rapidement vers $\sqrt{2}$. C'est un exercice classique de mathématiques que de démontrer cette convergence. On peut par

exemple étudier la fonction $g : x \mapsto \dfrac{1}{2}\left(x + \dfrac{2}{x}\right)$. Elle stabilise $[\sqrt{2}, +\infty[$ et, sur cet intervalle, $g(x) \leqslant x$, avec égalité si et seulement si $x = \sqrt{2}$; etc.

Le fait que la dérivée de g en $x_0 = \sqrt{2}$ soit nulle est fondamental : c'est grâce à cela qu'on a une vitesse de convergence élevée. En effet, en posant $\delta_n = u_n - \sqrt{2}$, on a $\dfrac{\delta_{n+1}}{\delta_n} = \dfrac{g(u_n) - g(x_0)}{u_n - x_0} \underset{n\to+\infty}{\longrightarrow} g'(x_0)$, donc plus $|g'(x_0)|$ est faible, meilleure est la vitesse de convergence. Ici, on a même plus précisément $\delta_{n+1} = \dfrac{\delta_n^2}{2x_n} \leqslant \dfrac{\delta_n^2}{2}$; on dit que la convergence est quadratique, ou d'ordre 2.

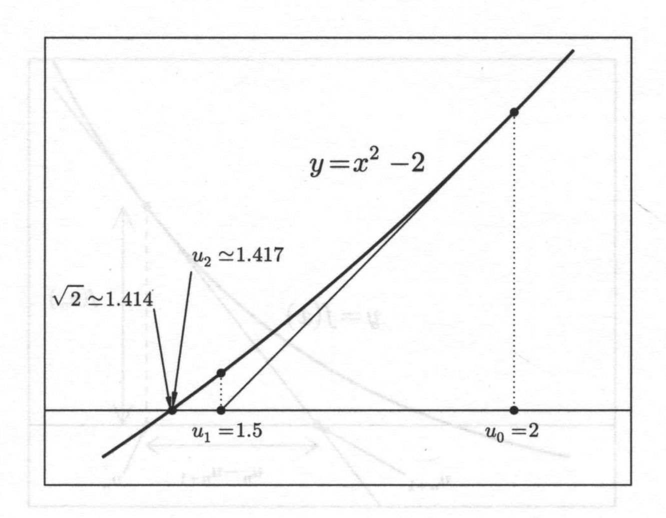


Figure 8.3
Les deux premières itérations du calcul approché de $\sqrt{2}$ par la méthode de Newton. Difficile de visualiser au delà !

Exercice 8.3 avec corrigé Avec les majorations précédentes, à partir de quel n est-on certain d'avoir $|\delta_n| \leqslant \dfrac{1}{2^{1000}}$ (autrement dit les 1 000 premiers bits significatifs de $\sqrt{2}$) ?

On a $\delta_1 \leqslant \dfrac{1}{2}$, donc $0 \leqslant \delta_2 \leqslant \dfrac{1}{2^2}$ (on a laissé de côté un facteur $\dfrac{1}{2}$), puis $0 \leqslant \delta_3 \leqslant \dfrac{1}{2^4}$, puis par récurrence immédiate : $0 \leqslant \delta_n \leqslant \dfrac{1}{2^{n-1}}$ pour tout $n \geqslant 1$. Pour avoir $|\delta_n| \leqslant \dfrac{1}{2^{1000}}$, il suffit donc d'avoir $2^{n-1} \geqslant 1000$, c'est-à-dire $n \geqslant 11$. Relisez la conclusion : en 11 itérations, on obtient 1 000 bits significatifs ; spectaculaire, non ?

Cet algorithme d'extraction de racine était probablement connu des Babyloniens (voir en fin de chapitre l'exercice 8.16), même s'il n'était évidemment pas question de dérivée, mais plutôt d'un raisonnement géométrique.

8.2.2 Algorithme général, terminaison, correction et complexité

L'algorithme général est celui décrit dans le cas particulier de la section précédente : on part d'une première valeur et on suit la tangente ; on continue avec l'intersection de cette tangente avec l'axe des abscisses (figure 8.4) jusqu'à ce que la différence entre deux termes consécutifs soit assez petite. Il est nécessaire (on croise les doigts) de ne jamais rencontrer de point en lequel la dérivée de f s'annule.

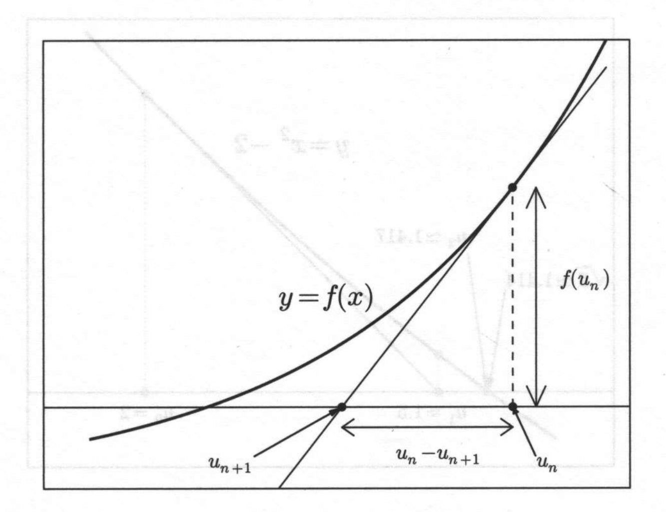


Figure 8.4
Dans la méthode de Newton, $f'(u_n) = \dfrac{f(u_n)}{u_n - u_{n+1}}$, donc $u_{n+1} = u_n - \dfrac{f(u_n)}{f'(u_n)}$.

Données : f, g, u_0, ε *(g représente f')*
$u \leftarrow u_0$
$v \leftarrow u - f(u)/g(u)$
tant que $|v - u| > \varepsilon$ **faire**
  $u \leftarrow v$
  $v \leftarrow v - f(v)/g(v)$
Résultat : v

Avant même de parler de complexité d'un algorithme, il faut démontrer sa terminaison et sa correction, c'est-à-dire ici d'une part qu'il n'y aura pas de division par zéro, et d'autre part que le résultat renvoyé r sera tel qu'il existe x_0 zéro de f tel que $|x_0 - r| \leqslant \varepsilon$.

Or, cet algorithme est un cauchemar pour l'informaticien :

- On peut rencontrer des divisions par zéro.
- La terminaison n'est pas assurée.
- Même si un résultat est renvoyé, il peut être éloigné d'un zéro de f.

En pratique, les programmes réalisant la méthode de Newton sont donc plus complexes. Voici une condition *suffisante* simple qui assure la convergence (au moins mathématique, en faisant abstraction des erreurs de calcul) : si f est de classe $\mathcal{C}^1$, convexe sur un intervalle I, f possède un point d'annulation sur I et $u_0 \in I$ est tel que $f(u_0) > 0$, alors la méthode de Newton appliquée depuis le premier terme u_0 est convergente vers un zéro de f.

Cette condition peut sembler déraisonnablement compliquée. De fait, il ne faut pas espérer une vérification automatique d'une telle propriété. Cependant, pour une fonction raisonnable, la concavité/convexité locale est la règle : si f est de classe $\mathcal{C}^2$ avec $f''(x_0) \neq 0$, alors f'' est de signe strict constant au voisinage de x_0. Ainsi, si on part de u_0 assez proche d'un zéro vérifiant cette condition, alors la méthode de Newton convergera vers ce zéro.

Il est à noter enfin qu'on trouve parfois la condition d'arrêt $|u_{n+1} - u_n| \leqslant \varepsilon$ remplacée par une condition de la forme $|f(u_n)| \leqslant \varepsilon'$, ce qui est une condition de même nature... du moins si $f'(x_0) \neq 0$.

Exercice 8.4 Commenter cette condition d'arrêt, lorsque $f'(x_0) = f''(x_0) = 0$ et $f^{(3)}(x_0) \neq 0$.

8.2.3 Évaluation de la dérivée

La méthode de Newton nécessite la connaissance de la dérivée de la fonction en jeu. Dans certains contextes, la fonction dérivée est connue a priori (si on souhaite programmer la méthode de Newton de façon non générique, pour une fonction bien déterminée). Dans d'autres contextes tels que le calcul formel, on a la possibilité de calculer une expression de f' à l'aide de celle de f.

On peut aussi vouloir appliquer la méthode de Newton avec la simple connaissance de f, connue via ses *valeurs* données par une fonction, et non via une *expression*. Il convient alors d'approximer les valeurs de f' au mieux.

Une première façon raisonnable de le faire consiste à écrire : $f'(x) \simeq \dfrac{f(x+h) - f(x)}{h}$, avec h différent de zéro mais assez petit. Le théorème de Taylor-Young assure que si f est deux fois dérivable en x_0 et $f''(x_0) \neq 0$, alors :

$$\frac{f(x_0+h) - f(x_0)}{h} - f'(x_0) \sim \frac{f''(x_0)}{2} h.$$

Toutefois, on peut gagner très facilement un ordre : si f est de classe C^3 et $f^{(3)}(x_0) \neq 0$, alors :

$$\frac{f(x_0+h)-f(x_0-h)}{2h} - f'(x_0) \sim \frac{f^{(3)}(x_0)}{12}h^2.$$

Si $f^{(3)}(x_0) = 0$ on gagne évidemment encore un ordre. Ceci nous invite à approcher $f'(x_0)$ via le programme suivant :

```
def derivee(f, x0, h):
    return (f(x0+h) - f(x0-h)) / (2*h)
```

Dans l'exemple qui suit, on prend pour f la fonction exponentielle et on s'intéresse à des approximations de sa dérivée en 0, pour des valeurs de h allant de 1 à 2^{-30}. On représente figure 8.5 l'erreur $|\Delta_0(h) - 1|$ avec $\Delta_0(h) = \dfrac{f(h)-f(0)}{h}$, puis $\Delta_0(h) = \dfrac{f(h)-f(-h)}{2h}$. Les pentes sont bien celles espérées !

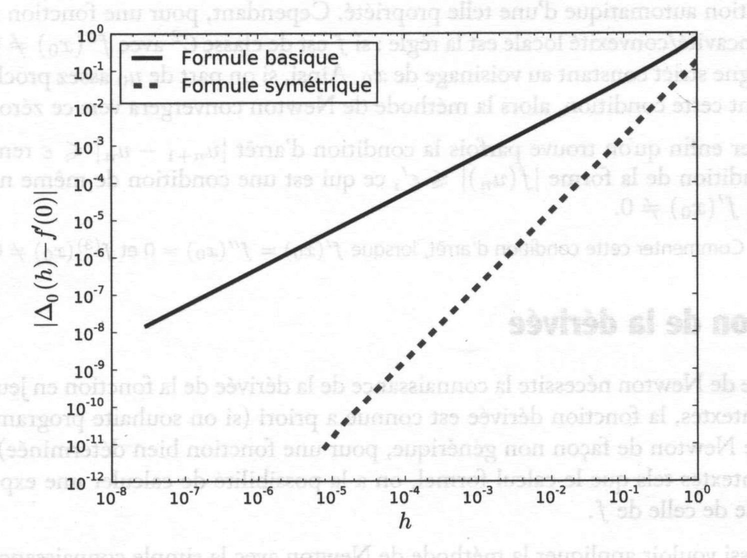


Figure 8.5
Comparaison entre les deux formules de dérivation discrète.

Exercice 8.5 À partir de $n = 26$ (respectivement 18), la méthode d'ordre 1 (respectivement 2) appliquée à $h = 2^{-n}$ renvoie le résultat « exact » `1.` jusqu'à $n = 52$ inclus, puis renvoie `0.` ; pourquoi ?
Avec la méthode symétrique d'ordre 2, on peut montrer que le pas optimal est de l'ordre de $\delta^{1/3}$, avec δ l'erreur relative dans l'évaluation de f (typiquement : de l'ordre de 2^{-b}, avec b le nombre de bits significatifs).

8.2.4 Mise en œuvre

Pour la version réalisée ici, on ne fait aucune vérification d'hypothèse de convergence, qu'on laisse à l'utilisateur. La terminaison et la correction ne sont donc pas assurées.

PROGRAMME 6 Méthode de Newton

```
def newton(f, fp, x0, epsilon):
    u = x0
    v = u - f(u)/fp(u)
    while abs(v-u) > epsilon:
        u, v = v, v - f(v)/fp(v)
    return v
```

Dans des conditions réelles, à défaut de pouvoir s'assurer de la convergence, on vérifierait à la volée que les valeurs prises par u et v ne sont pas trop grandes (ce qui signifierait qu'on quitte le domaine dans lequel l'action est censée se passer) et/ou que celles prises par $f'(v)$ ne sont pas trop petites.

```
In [3]: newton(math.sin, math.cos, 3, 10**(-3))
Out[3]: 3.141592653589793
```

```
In [4]: newton(lambda x : x**2-2, lambda x : 2*x, 2., 10**(-2))
Out[4]: 1.4142156862745099
```

Exercice 8.6 Que se passe-t-il si dans le dernier appel on remplace `2.` par `2` ? Attention, la réponse dépend de la version de Python !

En modifiant un peu le programme précédent, on peut imposer le nombre d'itérations au lieu de faire une boucle `while` :

```
In [5]: [newton_n_iterations(math.sin, math.cos, 3, k) for k in range(4)]
Out[5]: [3.0, 3.142546543074278, 3.141592653300477, 3.141592653589793]
```

```
In [6]: [newton_n_iterations(lambda x : x**2-2, lambda x : 2*x, 2., k) for k in range(5)]
Out[6]: [2.0, 1.5, 1.4166666666666667, 1.4142156862745099, 1.4142135623746899]
```

Enfin, on va visualiser l'évolution de l'erreur $|x_n - l|$ en fonction de n, le nombre d'itérations. On représente figure 8.6 cette erreur sur les deux exemples précédents (pour approcher $\sqrt{2}$ et π).

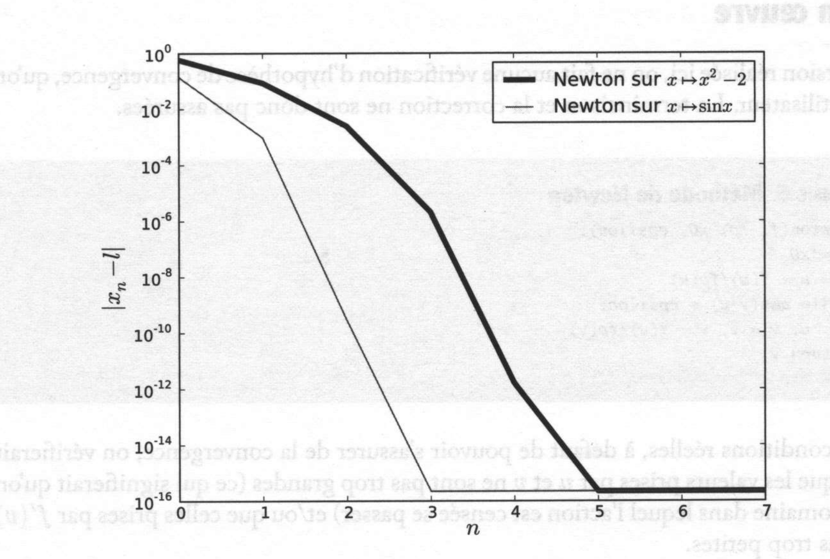


Figure 8.6
Vitesse de convergence de la méthode de Newton

8.3 Quelle méthode choisir ?

8.3.1 Bien cerner le contexte

On ne peut pas écrire une méthode universelle de résolution numérique d'équation de la forme $f(x) = 0$. Une fois que ceci est accepté, on peut chercher le meilleur compromis.

- Si on cherche la simplicité et la robustesse, la méthode dichotomique s'impose : la continuité de la fonction et un premier intervalle $[a, b]$ tel que $f(a)f(b) \leqslant 0$ sont suffisants. La lenteur de cette méthode est toute relative.
- Si on connaît une bonne approximation d'un zéro et si on cherche la rapidité, alors on peut appliquer une méthode de la famille de Newton : la méthode de Newton proprement dite si on connaît f' ou si on décide de l'approcher numériquement, la méthode de la sécante si on ne peut/veut pas évaluer f' (voir l'exercice 8.12) ou encore la méthode de Halley (qui est cubique) si on connaît f' et f''.
- Dans des situations intermédiaires, on peut pratiquer des méthodes mixtes, avec une première phase de localisation de zéros par dichotomie, puis des itérations de Newton. Si ces itérations de Newton ne semblent pas converger, on peut reprendre une phase de dichotomie, etc.

8.3.2 Utiliser numpy/scipy

La plupart du temps, il est déraisonnable de coder soi-même une méthode de résolution qui existe déjà et a probablement été plus testée et optimisée qu'une fonction « maison ».

Les bibliothèques `numpy` et `scipy.optimize` proposent les fonctions suivantes :

- La fonction `numpy.roots` détermine les racines d'un polynôme donné par la liste de ses coefficients.

```
In [7]: numpy.roots([1, 2, -1, -2])
Out[7]: array([1., -2., -1.])
```

  Elle fournit même les racines complexes, le nombre imaginaire i étant noté *j*.

```
In [8]: numpy.roots([1, 0, 0, 1])
Out[8]: array([-1.0 + 0.j, 0.5 + 0.8660254j, 0.5 - 0.8660254j])
```

```
In [9]: numpy.roots([1, 2, 2, 4])
Out[9]: array([-2.0e+00 + 0.j, 1.90013207e-16 + 1.41421356j, 1.90013207e-16 - 1.41421356j])
```

- La méthode dichotomique, qualifiée de lente mais sûre (*slow but sure*), est implémentée dans `scipy.optimize.bisect`.

```
In [10]: scipy.optimize.bisect(math.sin, 3, 4)
Out[10]: 3.141592653589214
```

- La méthode de Newton est programmée dans `scipy.optimize.newton`. Il est à noter que si on ne donne pas la dérivée, c'est en fait la méthode de la sécante qui est appliquée. Dans le cas opposé où on lui fournit f' et f'', c'est la méthode de Halley qui sera en fait mise en œuvre.

```
In [11]: scipy.optimize.newton(math.sin, 3, math.cos)
Out[11]: 3.141592653589793
```

- On dispose également de `scipy.optimize.brentq` qui implémente la méthode de Brent pour trouver une racine dans un intervalle donné. Le lecteur intéressé ira lire la documentation de cette fonction.

```
In [12]: scipy.optimize.brentq(math.sin, 3, 4)
Out[12]: 3.141592653589793
```

- Enfin, pour des problèmes qui ne sont pas scalaires, on peut utiliser la fonction `fsolve` :

```
In [13]: scipy.optimize.fsolve(lambda (x,y): (x + y**2, 1 - y + x**2), (0,0))
Out[13]: array([-0.33145847, 0.75398698])
```

Lire la documentation de ces fonctions est assez intéressant. Il y est par exemple rappelé que rien n'est garanti dans le résultat.

Enfin, ces fonctions intégrées commentent parfois la qualité supposée de leur résultat :

```
In [14]: scipy.optimize.fsolve(lambda (x,y):(x+y**2,1+y+x**2),(0,0))
  /usr/lib/python2.7/dist-packages/scipy/optimize/minpack.py:152:
  RuntimeWarning: The iteration is not making good progress, as
  measured by the improvement from the last ten iterations.
  warnings.warn(msg, RuntimeWarning)
Out[14]: array([-0.33145845, -0.75398696])
```

8.4 Exercices

Exercice 8.7 Comment approcher numériquement $f''(x_0)$? Proposer une formule, puis une majoration du reste dans un cadre raisonnable. Tester la formule proposée sur des exemples simples et vérifier l'ordre de grandeur de l'erreur.

Dans les différents cas, dans quelle zone se trouvent les pas optimaux ?

Exercice 8.8 *Cet exercice est inspiré du chapitre 12 de [Holmgren], qui contient de nombreuses autres pistes d'expérimentation de la méthode de Newton sur les polynômes de petit degré.*

On s'intéresse ici aux valeurs obtenues par itération de la méthode de Newton pour la fonction $f : x \mapsto x^3 + cx + 1$, avec 0 comme valeur initiale.

1 Pour $c > 0$, la méthode de Newton converge pour tout choix de la valeur initiale. Faire un dessin pour s'en convaincre et démontrer effectivement le résultat à titre d'exercice de mathématiques. Expérimenter avec différentes valeurs de c et u_0.

2 On prend ici $c = 0$. Donner les premiers termes, si $u_0 = 0.8$. À l'aide d'un dessin, prédire le comportement de la méthode de Newton en fonction du premier terme. Expérimenter et démontrer.
On pourra établir que la méthode de Newton est convergente, sauf pour un ensemble dénombrable de valeurs initiales, pour lesquelles la suite des itérées rencontre la valeur 0 et n'est alors plus définie au delà.

3 Dans cette question, on fixe $c = -1$. La méthode de Newton est-elle convergente ?

Sur la figure 8.7, c prend 2000 valeurs entre -1.29 et -1.27 ; on représente (avec un point) chaque itérée u_n de Newton avec $u_0 = 0$ et $100 \leqslant n \leqslant 150$. Plus précisément, on ne représente que celles appartenant à $[-0.15, 0.1]$: il existe une autre zone, vers $u_0 = 0.8$, dans laquelle on observe le même schéma de bifurcation.

Exercice 8.9 Dans la méthode d'extraction de racines du paragraphe 8.2.1, si on remplace la relation $u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right)$ par $u_{n+1} = \frac{1}{2}\left(u_n + \frac{K}{u_n}\right)$ où K est un *réel positif*, alors la suite $(u_n)_{n\in\mathbb{N}}$ converge vers $\sqrt{K}$.

1 Appliquer la méthode de Newton à la fonction $x \mapsto x^2 - K$, avec K le complexe $1 + i$ et différentes valeurs initiales u_0.
Le complexe $1 + i$ est construit en Python par `complex(1,1)` : aucune bibliothèque n'est requise.

2 Comparer le résultat à celui renvoyé par `complex(1,1)**(0.5)`.

3 Question de mathématiques : en fonction du premier terme, vers quelle « racine carrée » de $z_0 \neq 0$ converge la méthode de Newton appliquée à $z \mapsto z^2 - z_0$?

Exercice 8.10 * Si A est une matrice symétrique réelle définie positive, alors on peut montrer que la suite définie par $M_0 = I_n$ et la relation de récurrence $M_{p+1} = \frac{1}{2}\left(M_p + AM_p^{-1}\right)$ est bien définie, et que ses termes sont des matrices symétriques définies positives convergeant vers B qui est *la* matrice symétrique réelle définie positive de carré égal à A.

1 Trouver une condition d'arrêt raisonnable pour transformer ce résultat de mathématiques en algorithme effectif.
2 Programmer cet algorithme. Évaluer sa complexité.
3 Tester le programme sur des matrices définies positives, telles que celles de Virginie puis de Hilbert, définies au chapitre précédent.
4 Démontrer, à titre d'exercice de mathématiques, le résultat admis dans l'énoncé.

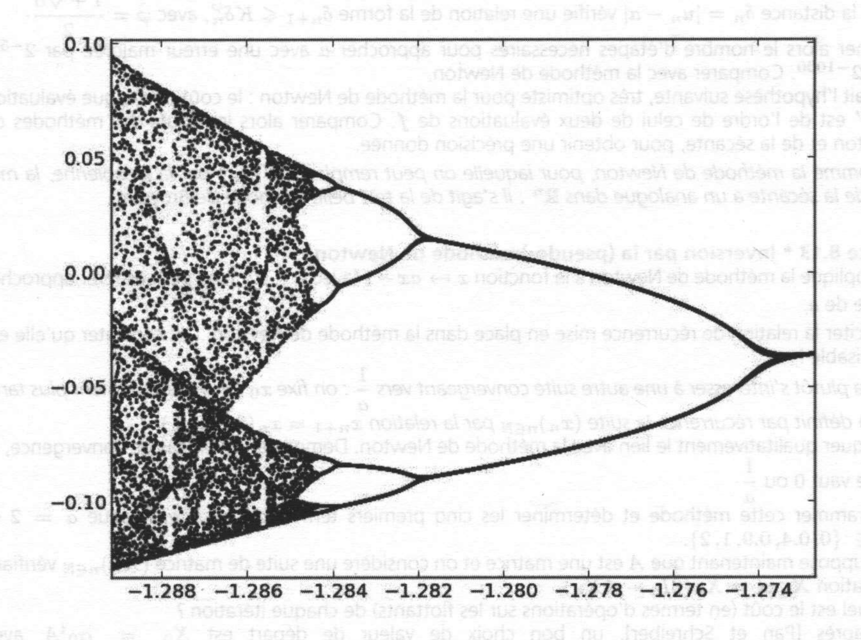


Figure 8.7
Diagramme de bifurcation : le chaos s'installe...

Exercice 8.11 * Si on applique la méthode de Newton à l'application complexe $z \mapsto z^3 - 1$, on obtient une suite qui converge (sauf pour un ensemble dénombrable de valeurs initiales) vers 1, $j = e^{2i\pi/3}$ ou j^2. Vérifier expérimentalement cette affirmation.

Le lecteur courageux construisant un graphique représentant pour chaque valeur initiale la valeur limite à l'aide d'un point d'une certaine couleur... sera bien récompensé de ses efforts ! Les ensembles représentés s'appellent des bassins d'attraction et sont ici des fractals.

Exercice 8.12 Méthode de la sécante.
Soit $f : I \to \mathbb{R}$ continue, s'annulant en $a \in I$, concave ou convexe au voisinage de a. On fixe $u_0, u_1 \in I$ au voisinage de a et on construit $(u_n)_{n \geqslant 2}$ de la façon suivante : pour tout $n \geqslant 2$, on définit u_n comme l'abscisse de l'intersection de l'axe des abscisses avec la sécante au graphe de f passant par les points d'abscisse u_{n-2} et u_{n-1}.

1 Faire un dessin ; constater qu'il peut y avoir divergence, ou même que u_n peut ne pas être défini à partir d'un certain rang, mais que si u_0 et u_1 sont assez proches de a, on peut raisonnablement espérer qu'il y ait convergence de $(u_n)_{n \in \mathbb{N}}$ vers a.
2 Donner une relation liant u_{n-2}, u_{n-1} et u_n.
3 Proposer un test d'arrêt pour la méthode de la sécante, qui consiste à approcher a en calculant des u_n jusqu'à ce qu'une certaine condition soit vérifiée.
4 Programmer et tester la méthode de la sécante.
5 On admet que sous des conditions favorables (mais réalistes si u_0 et u_1 sont suffisamment proches de a), alors la distance $\delta_n = |u_n - a|$ vérifie une relation de la forme $\delta_{n+1} \leqslant K\delta_n^{\varphi}$, avec $\varphi = \dfrac{1+\sqrt{5}}{2}$.
Donner alors le nombre d'étapes nécessaires pour approcher a avec une erreur majorée par 2^{-52}, puis 2^{-1000}. Comparer avec la méthode de Newton.
6 On fait l'hypothèse suivante, très optimiste pour la méthode de Newton : le coût de chaque évaluation de f' est de l'ordre de celui de deux évaluations de f. Comparer alors les coûts des méthodes de Newton et de la sécante, pour obtenir une précision donnée.

Tout comme la méthode de Newton, pour laquelle on peut remplacer $f'(u_n)$ par la jacobienne, la méthode de la sécante a un analogue dans $\mathbb{R}^n$: il s'agit de la très belle méthode de Broyden.

Exercice 8.13 * Inversion par la (pseudo-)méthode de Newton.
Si on applique la méthode de Newton à la fonction $x \mapsto ax - 1$ (avec $a \in \mathbb{R}_+^*$), on peut espérer approcher l'inverse de a.

1 Expliciter la relation de récurrence mise en place dans la méthode de Newton... et constater qu'elle est inutilisable !
On va plutôt s'intéresser à une autre suite convergeant vers $\dfrac{1}{a}$: on fixe x_0 (on verra comment plus tard) et on définit par récurrence la suite $(x_n)_{n \in \mathbb{N}}$ par la relation $x_{n+1} = x_n(2 - ax_n)$.
2 Expliquer qualitativement le lien avec la méthode de Newton. Démontrer qu'en cas de convergence, la limite vaut 0 ou $\dfrac{1}{a}$.
3 Programmer cette méthode et déterminer les cinq premiers termes de la suite, lorsque $a = 2$ et $x_0 \in \{0, 0.4, 0.9, 1, 2\}$.
4 On suppose maintenant que A est une matrice et on considère une suite de matrice $(X_n)_{n \in \mathbb{N}}$ vérifiant la relation $X_{n+1} = X_n(2I_n - AX_n)$.
a) Quel est le coût (en termes d'opérations sur les flottants) de chaque itération ?
b) D'après [Pan et Schreiber], un bon choix de valeur de départ est $X_0 = \alpha_0 {}^tA$, avec $\alpha_0 = \dfrac{1}{\|A\|_1 \|A\|_\infty}$, où $\|A\|_1 = \max_j \sum_i |a_{i,j}|$ et $\|A\|_\infty = \max_i \sum_j |a_{i,j}|$.
Programmer cette méthode d'inversion matricielle, en décidant d'un test d'arrêt raisonnable.
c) Évaluer la complexité de cette méthode (en cas de convergence, et en fonction du nombre d'itérations finalement effectuées).
d) Tester, comparer avec les méthodes du chapitre précédent.

On représente figure 8.8 l'évolution de $\left\|A^{-1} - X_n\right\|_1$ en fonction de n, avec $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}$.

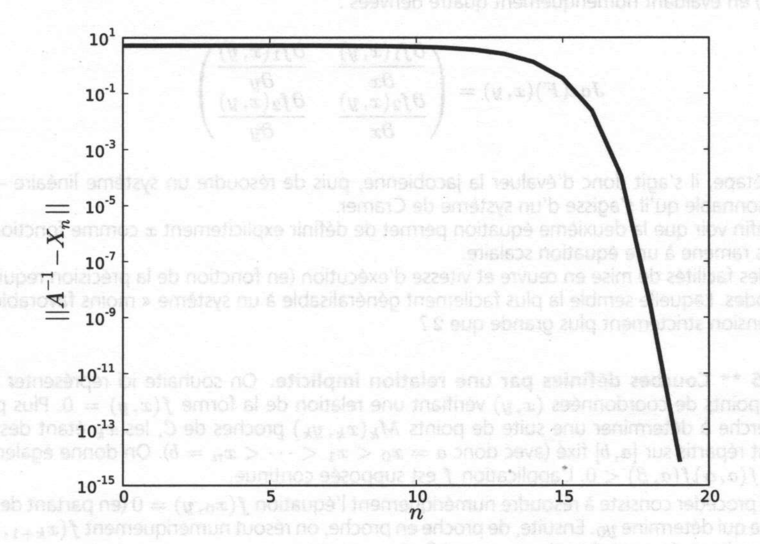


Figure 8.8
La convergence se fait désirer... puis est très rapide.

Exercice 8.14 ** On se propose d'approcher de trois façons différentes une solution du système non linéaire

$$\begin{cases} \exp(x + y^2)\sin(x + y) = 1 \\ \cos x + \sin^2 y = 1 \end{cases}$$

Plus précisément, on veut approcher l'unique solution appartenant à $[0, 1] \times [0, 1]$. Cet exercice peut constituer à lui seul une séance de TP complète, modulo quelques explications supplémentaires !

```
In [15]: scipy.optimize.fsolve(lambda (x,y):(f1(x,y),f2(x,y)),(0.5,0.5))
Out[15]: array([ 0.39348813, 0.28009462])
```

Dans chaque cas, on s'intéressera au nombre d'évaluations de f nécessaires pour obtenir p bits significatifs.

1 On peut, à x_0 fixé, utiliser une méthode de résolution approchant y tel que

$$e^{x_0+y^2}\sin(x_0 + y) = 1.$$

Si on note $\varphi(x_0)$ cet y, il reste à résoudre numériquement $\cos(x) + \sin^2(\varphi(x)) = 1$.

2 On peut adapter la méthode de Newton à la dimension 2 en construisant une suite $(X_n)_{n\in\mathbb{N}}$ de couples de réels tels que $X_0 = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$ par exemple, avec la relation

$$X_{n+1} = X_n - (dF_{X_n})^{-1}.F(X_n)$$

où $F : (x, y) \mapsto (f_1(X), f_2(X)) = \left(\exp(x + y^2)\sin(x + y) - 1, \cos x + \sin^2 y - 1\right)$ est la fonction dont on cherche un zéro et dF_{X_n} est sa différentielle en X_n, dont on peut approcher la matrice (la

jacobienne) en évaluant numériquement quatre dérivées :

$$Jac(F)(x,y) = \begin{pmatrix} \dfrac{\partial f_1(x,y)}{\partial x} & \dfrac{\partial f_1(x,y)}{\partial y} \\ \dfrac{\partial f_2(x,y)}{\partial x} & \dfrac{\partial f_2(x,y)}{\partial y} \end{pmatrix}$$

À chaque étape, il s'agit donc d'évaluer la jacobienne, puis de résoudre un système linéaire — avec l'espoir raisonnable qu'il s'agisse d'un système de Cramer.

3 On peut enfin voir que la deuxième équation permet de définir explicitement x comme fonction de y, ce qui nous ramène à une équation scalaire.

4 Comparer les facilités de mise en œuvre et vitesse d'exécution (en fonction de la précision requise) des trois méthodes. Laquelle semble la plus facilement généralisable à un système « moins favorable » ? Et à une dimension strictement plus grande que 2 ?

Exercice 8.15 ** Courbes définies par une relation implicite. On souhaite ici représenter un ensemble $\mathcal{C}$ de points de coordonnées (x, y) vérifiant une relation de la forme $f(x, y) = 0$. Plus précisément, on cherche à déterminer une suite de points $M_k(x_k, y_k)$ proches de $\mathcal{C}$, les x_k étant des points uniformément répartis sur $[a, b]$ fixé (avec donc $a = x_0 < x_1 < \cdots < x_n = b$). On donne également α et β tels que $f(a, \alpha)f(a, \beta) < 0$. L'application f est supposée continue.

Une façon de procéder consiste à résoudre numériquement l'équation $f(x_0, y) = 0$ (en partant de l'intervalle $[\alpha, \beta]$), ce qui détermine y_0. Ensuite, de proche en proche, on résout numériquement $f(x_{k+1}, y) = 0$ en partant de valeurs de y proches de y_k, ce qui fixe y_{k+1}, etc.

1 Mettre en place cette idée dans un programme prenant en entrée $f, x_0, x_1, \alpha, \beta$ et n, et renvoyant la liste des x_k et celle des y_k.

2 En utilisant ce qui précède et la bibliothèque **matplotlib** (voir chapitre 9), représenter sur un même graphique les courbes d'équations implicites $\exp(x + y^2)\sin(x + y) = 1$ et $\cos x + \sin^2 y = 1$ (tirées de l'exercice précédent).

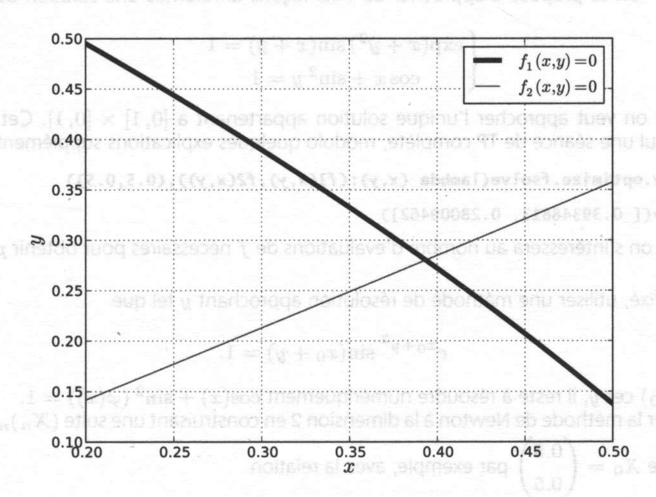


Figure 8.9
L'intersection se situe vers $(0.39, 0.28)$.
Au voisinage de $(0, 0)$, **l'équation** $f_2(x, y) = 1$ **pose problème ; pourquoi ?**

Exercice 8.16 Les Babyloniens et $\sqrt{2}$.

1 Donner les quatre premiers chiffres significatifs de $\sqrt{2}$... en base 60.
2 Déchiffrer rapidement la tablette YBC 7289 représentée figure 8.10 et dont l'âge est estimé à plus de 3 000 ans.
3 Constater que les Babyloniens connaissaient un peu de mathématiques !



Figure 8.10
La tablette YBC7289.
On notera un certain manque de soin dans le tracé des lignes droites.

9

Résolution numérique d'équations différentielles

Hormis quelques cas d'école simples, on ne sait pas déterminer d'expressions analytiques pour les solutions d'équations différentielles. Le but de ce chapitre est de présenter la méthode d'Euler, qui sert à en calculer des approximations.

De nombreux phénomènes physiques se modélisent à l'aide d'équations différentielles pour lesquelles on ne dispose pas de solutions analytiques : un pendule amorti nous amène à étudier l'équation $\ddot{\theta} = -k_1 \sin\theta - k_2\dot{\theta}$, les problèmes de cinétique chimique conduisent à des systèmes différentiels *non linéaires* décrivant les évolutions de différents réactifs au cours du temps, et les phénomènes qu'on observe en mécanique des fluides sont en partie décrits par les équations aux dérivées partielles non linéaires de type *Navier-Stokes*.

En mathématiques, ces équations ont leur intérêt propre et étudier le comportement qualitatif de solutions est nettement plus aisé si on peut visualiser une approximation raisonnable de celles-ci.

9.1 Méthode d'Euler

Le théorème de Cauchy-Lipschitz assure[1] que sous des conditions raisonnables, il existe une unique application y de classe $\mathcal{C}^1$ sur $[a, b]$ dont la valeur est imposée en a et qui vérifie une équation de la forme $y'(t) = F(t, y(t))$ pour tout $t \in [a, b]$. L'objet des *schémas numériques* est d'obtenir des approximations de ces solutions dont la théorie donne l'existence de façon non constructive. En pratique, on tente en général d'approcher y en un certain nombre de points répartis sur l'intervalle $[a, b]$.

9.1.1 Principe de la méthode d'Euler

Il s'agit de calculer une approximation y_k des $y(t_k)$, avec $t_k = a + kh$, où $h = \dfrac{b-a}{n}$ est un *pas* qu'il conviendra d'ajuster. De façon très simple, si on écrit :

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(u)du = \int_{t_k}^{t_{k+1}} F(u, y(u))du \simeq hF(t_k, y(t_k)),$$

alors on obtient la méthode d'Euler : les approximations sont calculées de proche en proche via la formule suivante : $y_{k+1} = y_k + hF(t_k, y_k)$ (voir figure 9.1). On initialise bien entendu avec $y_0 = y(a)$, qui sera la seule valeur « exacte » calculée.

Graphiquement, cela revient à faire des approximations successives de courbes par des tangentes. On va prendre pour exemple l'équation différentielle $y'(t) = y(t)$ (dont les solutions sont les $t \mapsto K\mathrm{e}^t$) et on va approcher sur $[0, 1]$ l'unique solution telle que $y(0) = 1$, autrement dit la fonction exponentielle. Pour $n = 3$, c'est-à-dire un pas de $\dfrac{1}{3}$, on obtient la figure 9.2.

1. Comme vu en cours de mathématiques, c'est évidemment plus compliqué. Par exemple, si F n'est pas linéaire vis-à-vis de y, on est juste assuré de l'existence d'une solution dont la valeur en a est imposée, mais on ne sait pas si elle est définie jusqu'en b.

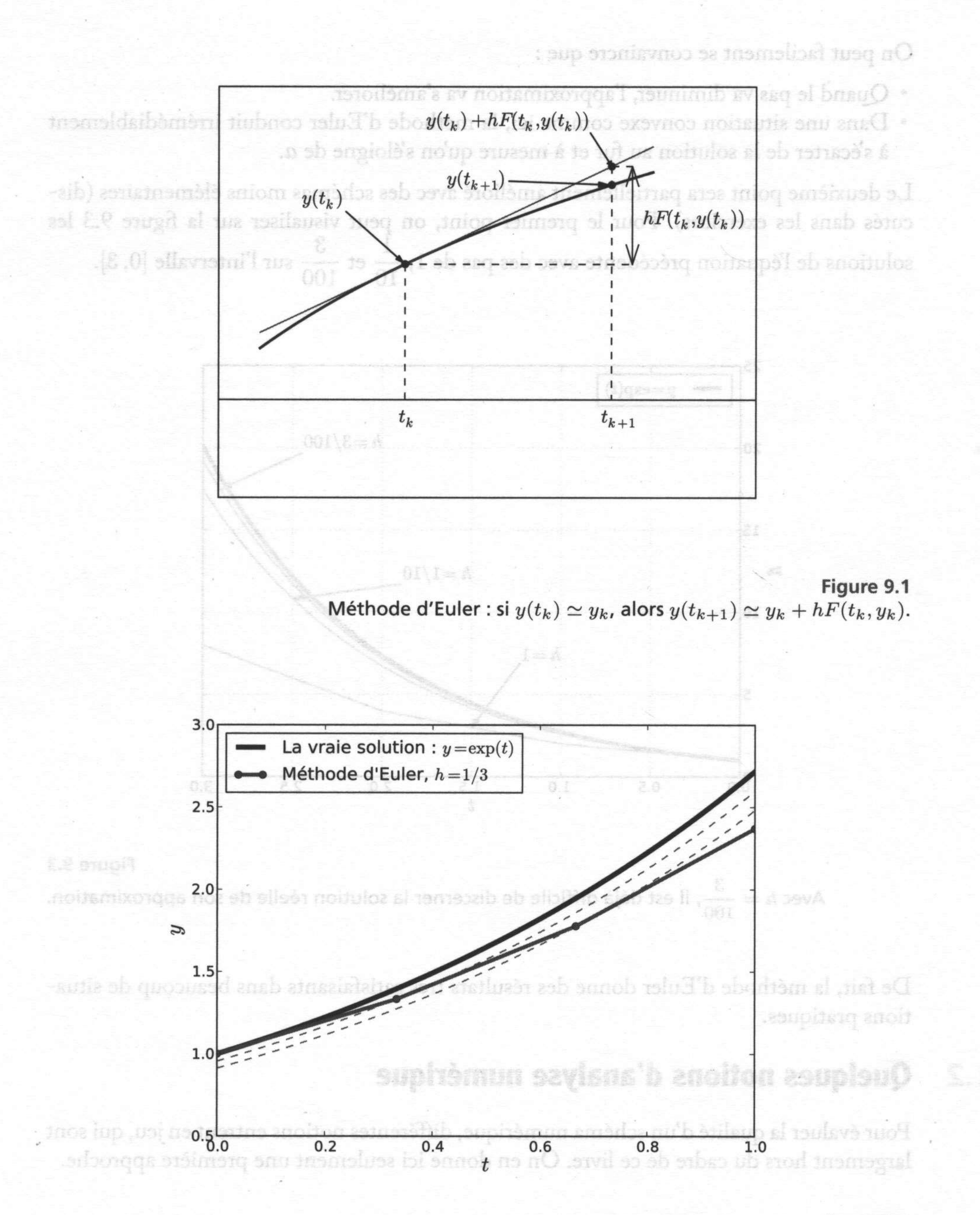


Figure 9.1
Méthode d'Euler : si $y(t_k) \simeq y_k$, alors $y(t_{k+1}) \simeq y_k + hF(t_k, y_k)$.

Figure 9.2
En pointillés : les fonctions exponentielles correspondant aux nouvelles conditions initiales, après chaque itération. Pour l'étape ultérieure, on suit la tangente.

On peut facilement se convaincre que :

- Quand le pas va diminuer, l'approximation va s'améliorer.
- Dans une situation convexe comme ici, la méthode d'Euler conduit irrémédiablement à s'écarter de la solution au fur et à mesure qu'on s'éloigne de a.

Le deuxième point sera partiellement amélioré avec des schémas moins élémentaires (discutés dans les exercices). Pour le premier point, on peut visualiser sur la figure 9.3 les solutions de l'équation précédente avec des pas de 1, $\frac{1}{10}$ et $\frac{3}{100}$ sur l'intervalle $[0, 3]$.

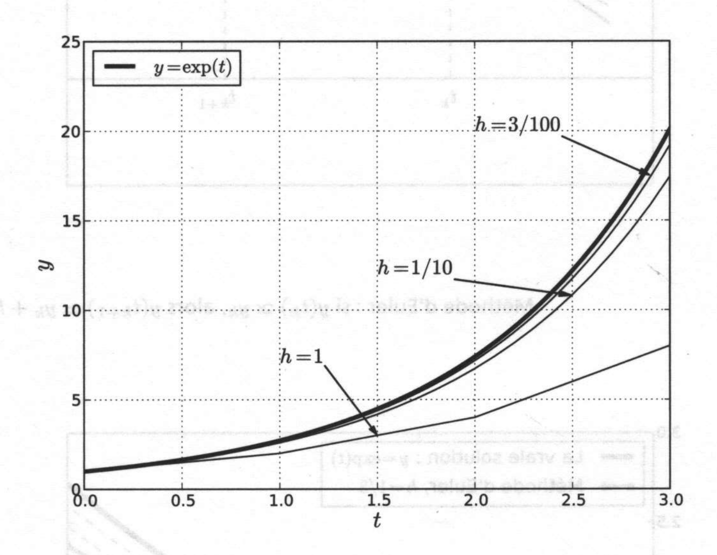


Figure 9.3

Avec $h = \frac{3}{100}$, il est déjà difficile de discerner la solution réelle de son approximation.

De fait, la méthode d'Euler donne des résultats très satisfaisants dans beaucoup de situations pratiques.

9.1.2 Quelques notions d'analyse numérique

Pour évaluer la qualité d'un schéma numérique, différentes notions entrent en jeu, qui sont largement hors du cadre de ce livre. On en donne ici seulement une première approche.

On va supposer qu'on approche une solution y sur un intervalle $[a, b]$ avec une méthode à un pas telle que la méthode d'Euler. À h fixé, on note $n(h)$ le nombre de temps $t_i^{(h)}$ en lesquels y sera approchée (environ $\frac{b-a}{h}$). On aimerait avoir : $y(t_i^{(h)}) \simeq y_i^{(h)}$, avec $y_i^{(h)}$ la valeur obtenue dans le schéma de pas h.

Le premier objectif est d'avoir une méthode *convergente* au sens suivant :

$$\max_{0\leqslant i\leqslant n(h)} \left| y\left(t_i^{(h)}\right) - y_i^{(h)} \right| \underset{(y_0,h)\to(y(a),0)}{\longrightarrow} 0.$$

On aura noté que la valeur de départ est supposée tendre vers $y(a)$; c'est important, car même si en théorie on pose $y_0^{(h)} = y(a)$, on peut s'attendre à avoir en pratique une valeur numérique seulement très proche.

La convergence sera assurée pour peu que le schéma soit :

- *stable* : en perturbant un peu le schéma, la perturbation résultante est contrôlée linéairement par la perturbation initiale[2] ;
- *consistant* : l'erreur de consistance, définie dans le cas de la méthode d'Euler par :

$$e(h) = \sum_{i=0}^{n(h)-1} |y(t_{i+1}) - (y(t_i) + hF(t_i, y(t_i)))|$$

tend vers 0 lorsque h tend vers 0. Cela signifie que si à chaque étape on remplace y_i par $y(t_i)$ et si on somme les erreurs provoquées par les approximations locales, alors l'erreur globale produite reste contrôlée par h.

On dit qu'une méthode est d'ordre p lorsque l'erreur de consistance possède un majorant de la forme $K.h^p$. Plus précisément, l'ordre d'une méthode sera le maximum des entiers p pour lesquels on a une majoration de cette forme.

On pourra retenir pour la suite que :

- La méthode d'Euler est d'ordre 1. On parle ici de la méthode d'Euler *explicite* et non de sa variante, la méthode d'Euler implicite.
- La méthode de Heun est d'ordre 2 (voir plus loin l'exercice 9.7).
- La méthode dite de Runge-Kutta (ou « Runge-Kutta d'ordre 4 », voir plus loin l'exercice 9.8) est d'ordre... devinez !

Concrètement, on peut visualiser ces ordres en représentant les erreurs de consistance $e(h)$ en fonction de h avec une échelle doublement logarithmique (voir la figure 9.4).

2. Pour plus de précisions, consulter tout cours d'analyse numérique.

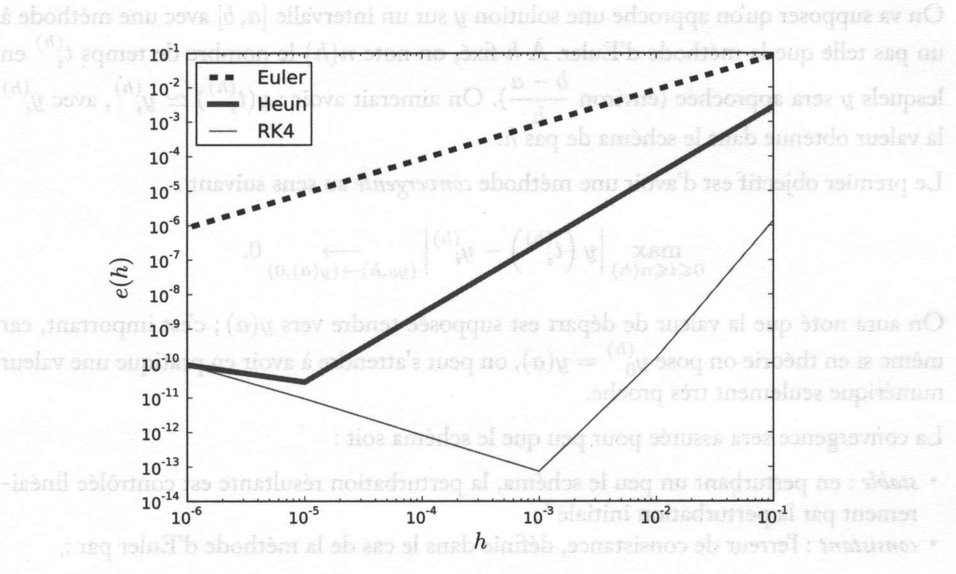


Figure 9.4
Erreurs de consistance : h en abscisse et $e(h)$ en ordonnée pour $y' = y$ et $y(0) = 1$ sur $[0, 1]$.

Exercice 9.1 Montrer que pour la méthode d'Euler appliquée à $y' = y$ sur $[0, 1]$ avec $y(0) = 1$ et $h = \dfrac{1}{n}$, on a $e(h) \sim \dfrac{\mathrm{e} - 1}{2n}$.

Exercice 9.2 Expliquer le comportement des courbes de la figure 9.4 sachant qu'on a calculé les erreurs de consistance en sommant effectivement $n = \dfrac{1}{h}$ termes.

9.1.3 Choix du pas

Le choix du pas est une étape obligée lors de la mise en place d'une méthode numérique de résolution.

- Si on choisit ce pas trop petit, le temps de calcul sera élevé.
- Si au contraire h est trop grand, l'erreur de consistance sera trop importante.

Il s'agit donc, comme toujours en calcul scientifique, de faire un compromis entre le temps de calcul et la qualité de l'approximation, et ce n'est pas toujours simple.

On s'intéresse maintenant aux temps de calcul nécessaires pour résoudre numériquement $y' = y$ sur $[0, 1]$, avec la condition initiale $y(0) = 1$, pour différents pas $h = \dfrac{1}{n}\cdot$ On évalue par ailleurs l'erreur $E(h) = \left|\mathrm{e} - y_n^{(h)}\right|$. Une bonne mesure de l'erreur est :

$$E(h) = \max_{0 \leqslant i \leqslant n} \left| y(t_i^{(h)}) - y_i^{(h)} \right|,$$

mais ici ce maximum est pris en $t_n = 1$, donc il suffit d'évaluer $\left|y(1) - y_n^{(h)}\right|$.

Les temps sont exprimés en secondes et on donne deux chiffres significatifs :

n	10^3	10^4	10^5	10^6	10^7
Temps	6.7×10^{-4}	6.9×10^{-3}	6.8×10^{-2}	6.7×10^{-1}	6.7
Erreur	1.36×10^{-3}	1.36×10^{-4}	1.36×10^{-5}	1.36×10^{-6}	1.36×10^{-7}

Les résultats sont sans surprise : d'une part, le temps est proportionnel au nombre de termes calculés et, d'autre part, on a ici $y_i^{(h)} = (1 + 1/n)^i$, donc :

$$E(1/n) = \mathrm{e} - (1 + 1/n)^n \sim \frac{\mathrm{e}}{2n}.$$

On aura noté que cette erreur est supérieure à celle de consistance, ce qui est bien normal.

Avec la méthode d'Euler, on ne peut obtenir une précision de l'ordre de 10^{-6} qu'avec un temps de calcul assez important. Pour ce type de précision (et mieux encore), on préfère donc les méthodes d'ordre 2 ou plus. On obtient par exemple avec la méthode de Heun les résultats suivants, déjà nettement meilleurs :

n	10^3	10^4	10^5	10^6
Temps	1.4×10^{-3}	1.3×10^{-2}	1.3×10^{-1}	1.4
Erreur	4.5×10^{-7}	4.5×10^{-9}	4.5×10^{-11}	4.1×10^{-13}

Enfin, avec la méthode de Runge-Kutta d'ordre 4, il n'est pas utile de prendre h très petit pour avoir une erreur spectaculairement faible avec un temps de calcul très court :

n	10^2	10^3	10^4
Temps	5.9×10^{-5}	3.9×10^{-4}	3.4×10^{-3}
Erreur	2.1×10^{-6}	2.2×10^{-10}	2.0×10^{-14}

SAVOIR-FAIRE Choisir le pas d'une méthode

Il faut connaître l'*ordre* de la méthode utilisée, la *précision souhaitée* et *le temps de calcul jugé acceptable*. Si on ne trouve pas de compromis avec ces données, on prend une méthode d'ordre supérieur.

POUR ALLER PLUS LOIN Pas adaptatif et méthodes multipas

Les programmes chargés d'intégrer numériquement des équations différentielles (tels que `odeint`) adaptent en général leur pas à la fonction F : si $F(t_k, y_k)$ est faible, alors y varie peu, donc on peut faire de plus grands pas. On parle de méthodes à *pas adaptatif*, ou *variable*.

Il existe également des méthodes dites *à pas multiples*, ou *multipas* : on calcule y_k à l'aide de $y_{k-1}, y_{k-2}, \dots y_{k-p}$. La méthode d'Adams est la plus classique de ces méthodes.

9.2 Mise en œuvre

On met ici en place la méthode d'Euler, mais la démarche est très proche pour d'autres méthodes à un pas. On pourra consulter les exercices 9.7 et 9.8 à ce sujet.

9.2.1 Équations scalaires d'ordre 1

La fonction `euler` va prendre en entrée une fonction F, les bornes a et b de l'intervalle d'étude, la condition initiale y_0 et le pas h. Plus précisément : avec ces données, la fonction va déterminer les approximations de la solution de $y'(t) = F(t, y(t))$ avec la condition initiale $y(a) = y_0$, en rendant un tableau de temps et un tableau de valeurs approchées par la méthode d'Euler. Les temps sont les $a + kh$ majorés par b.

Ces tableaux sont construits à l'aide de listes auxquelles on adjoint les nouveaux termes calculés. Cette façon de procéder est économe (on ne crée pas le tableau dès le début) et présente une bonne complexité (voir le chapitre 12).

PROGRAMME 7 Méthode d'Euler

```
def euler(F, a, b, y0, h):
    y = y0
    t = a
    les_y = [y0] # la liste des valeurs renvoyées
    les_t = [a]
    while t+h <= b:
        y += h * F(t, y)
        les_y.append(y)
        t += h
        les_t.append(t)
    return les_t, les_y
```

Un premier essai pour l'équation $y' = y$ avec la condition initiale $y(0) = 1$ et le pas $h = \dfrac{1}{3}$ donne ce qui suit.

```
In [1]: euler(lambda t, y :y, 0, 1, 1, 1./3)[1]
Out[1]: [1, 1.3333333333333333, 1.7777777777777777, 2.3703703703703702]
```

Au lieu d'utiliser une fonction anonyme, on aurait pu définir avant :

```
def f0(t, y):
    return y
```

puis faire l'appel suivant :

```
In [2]: euler(f0, 0, 1, 1, 1./3)[1]
Out[2]: [1, 1.3333333333333333, 1.7777777777777777, 2.3703703703703702]
```

On peut s'intéresser à la valeur approchée en 1 pour des pas de la forme 10^{-k} :

```
In [3]: [euler(lambda t, y :y, 0, 1, 1, 1./10**k)[1][-1] for k in range(1,5)]
Out[3]: [2.5937424601, 2.6780334944767583, 2.7142097225133828, 2.7181459268252266]
```

On aura noté l'expression typique de Python `t[-1]`, qui donne le dernier élément d'un tableau. Pratique certes, mais il ne faut pas abuser de ce genre de choses.

Dans ce dernier exemple, on résout $y' = y^2$ avec $y(1) = 2$. La solution $(y : t \mapsto \dfrac{1}{2-t})$ explose en temps fini :

```
In [4]: [euler(lambda t, y :y**2, 1, 2, 1, 10**(-k))[1][-1] for k in range(5)]
Out[4]: [2, 4.289186403020769, 24.424229986946074, 193.13676042981598, 1393.599920900794]
```

On peut visualiser figure 9.5 les solutions renvoyées pour les pas 10^{-k}, avec $k \in [\![1, 4]\!]$.

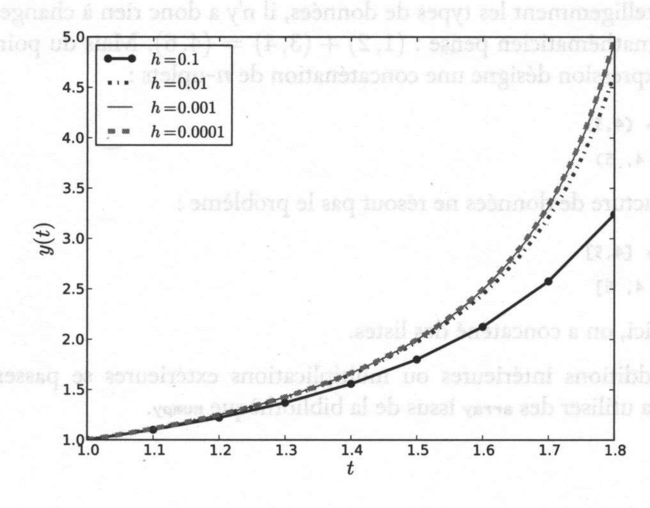


Figure 9.5
Au voisinage de ce type de point, le pas doit être plus petit.

9.2.2 Équations scalaires d'ordre 2 ou plus

Une équation scalaire telle que $\ddot{\theta} = -k \sin\theta$ est d'ordre 2 (ici, l'ordre désigne la plus grande dérivée intervenant dans l'équation, il n'est pas question de l'ordre d'un schéma numérique). Si on s'intéresse au couple $X = (\theta, \dot{\theta})$, alors il vérifie l'équation $\dot{X} = G(X)$, avec G l'application[3] $(\alpha, \beta) \mapsto (\beta, -k \sin\alpha)$. On ramène par ce procédé de *vectorisation* les équations différentielles scalaires d'ordre p à des équations d'ordre 1, mais à valeurs dans $\mathbb{R}^p$.

Par chance, la méthode d'Euler (mais aussi les autres méthodes numériques usuelles) fonctionne aussi bien pour des fonctions à valeurs vectorielles que réelles ! Il n'y a donc pas grand-chose à modifier :

```
def euler_vectoriel(F, a, b, y0, h):
    y = y0
    t = a
    les_y = [y0] # la liste des valeurs renvoyées
    les_t = [a]
    while t+h <= b:
        y = y + h * F(t, y)
        les_y.append(y)
        t += h
        les_t.append(t)
    return les_t, les_y
```

Exercice 9.3 avec corrigé Trouver les différences avec la version scalaire.
Regardez le nom de la fonction : il a changé ! Mais il n'y a pas que cela...

Si on choisit intelligemment les types de données, il n'y a donc rien à changer. Attention cependant. Le mathématicien pense : $(1,2) + (3,4) = (4,6)$. Mais du point de vue de Python, cette expression désigne une concaténation de n-uplets :

```
In [5]: (1,2) + (4,5)
Out[5]: (1, 2, 4, 5)
```

Changer de structure de données ne résout pas le problème :

```
In [6]: [1,2] + [4,5]
Out[6]: [1, 2, 4, 5]
```

Encore perdu : ici, on a concaténé des listes.

Pour que les additions intérieures ou multiplications extérieures se passent « comme dans $\mathbb{R}^n$ », on va utiliser des `array` issus de la bibliothèque `numpy`.

3. L'expérience montre que chez le débutant, ne pas vouloir expliciter la fonction G (en donnant la valeur de $G(\alpha, \beta)$) est souvent une économie de temps très modeste et qui coûte très cher par la suite...

On importe cette dernière, par exemple via **from numpy import** *. Ensuite, on obtient le résultat souhaité :

```
In [7]: 2*array([1,2]) + array([3,4])
Out[7]: array([5, 8])
```

Exercice 9.4 avec corrigé * Si `t1` et `t2` sont des **array**, expliquer la différence entre `t1 += t2` et `t1 = t1 + t2`.

Tester sur un exemple, en définissant d'abord `t1` *et* `t2`*, ainsi que* `t3 = t1`*. L'opération* `t1 += t2` *ne va pas créer un nouveau tableau* **numpy (array)** *: elle va modifier le tableau existant, pointé par* `t1`*, mais aussi par* `t3`*. Si on exécute* `t1 = t1 + t2`*, un nouveau tableau est créé et* `t1` *pointe maintenant dessus, alors que* `t3` *pointe sur un tableau qui n'a pas changé.*

On teste le programme sur l'équation $y'' + y = 0$, avec les conditions initiales $(y(0), y'(0)) = (0, 1)$. Mathématiquement, la solution est connue — il s'agit de la fonction sin.

Le résultat est représenté figure 9.6. On voit que pour un pas $h = \frac{1}{100}$, les approximations conduisent à s'écarter de façon sensible de la véritable solution, puisqu'on quitte l'intervalle $[-1, 1]$. On peut alors diminuer le pas ou changer de méthode.

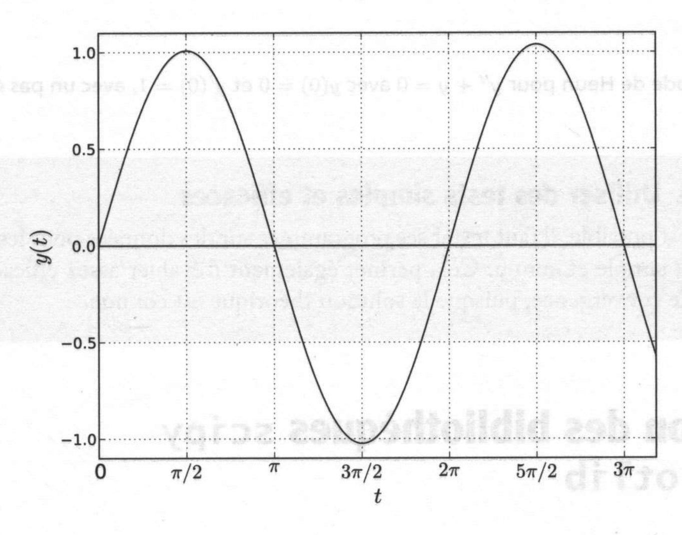


Figure 9.6
Méthode d'Euler pour $y'' + y = 0$ avec $y(0) = 0$ et $y'(0) = 1$, avec un pas égal à $\frac{1}{100}$

Sur la figure 9.7, on utilise le schéma de Heun, qui est d'ordre 2, avec le pas $\frac{1}{100}$; la différence avec la méthode d'Euler est nette.

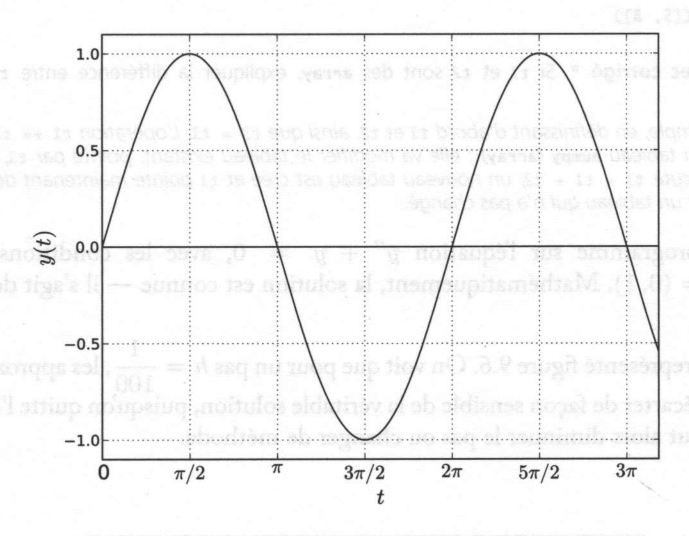


Figure 9.7
Méthode de Heun pour $y'' + y = 0$ avec $y(0) = 0$ et $y'(0) = 1$, avec un pas égal à $\frac{1}{100}$

SAVOIR-FAIRE Utiliser des tests simples et efficaces

Quand cela est possible, il faut tester ses programmes sur des données pour lesquelles le résultat est simple et connu. Cela permet également d'évaluer assez efficacement les qualités de convergence, puisque la solution théorique est connue.

9.3 Utilisation des bibliothèques `scipy` et `matplotlib`

SAVOIR-FAIRE Utiliser les bibliothèques de calcul standard pour résoudre un problème scientifique

On a montré dans les sections précédentes que les méthodes numériques exposées fonctionnaient vraiment et se programmaient assez facilement. Mais pour gagner en fiabilité, on utilise en général des fonctions fournies dans des bibliothèques dédiées.

9.3.1 Intégration des équations différentielles avec `odeint`

La bibliothèque `scipy.integrate` contient la fonction `odeint`, qui résout numériquement des équations différentielles. On commence donc par la charger :

```
from scipy.integrate import odeint
```

Une utilisation basique sera de la forme : `odeint(f, y0, t)`, pour résoudre l'équation $y'(t) = f(y(t), t)$ sur un intervalle $[a, b]$. L'aspect déroutant est la nature de t : il s'agit d'un tableau de temps entre a et b : `t` est de la forme `t=[a,t1,...,b]`. La condition initiale est alors : $y(a) = y_0$. La valeur renvoyée est un tableau contenant une estimation de la solution aux différents temps.

Attention Ordre des arguments

La fonction donnée en paramètre prend « le temps » comme deuxième argument, et non en premier comme on a l'habitude de le faire en mathématiques.

Par exemple, pour résoudre sur $[0, 1]$ l'équation $y' = y$ avec la condition initiale $y(0) = 1$ et un pas de 0.5, on peut exécuter :

```
In [8]: odeint(lambda y,t :y,1,[0,0.5,1])
Out[8]: array([[ 1. ], [ 1.64872127], [ 2.71828191]])
```

Il reste deux choses à apprendre :

- créer efficacement le tableau des temps en lesquels la fonction va être approchée ;
- représenter le graphe de la solution approchée.

Pour le premier point, on a trois possibilités :

1 À la main, on crée un tableau de valeurs, par exemple par compréhension :

```
In [9]: [k/10. for k in range(11)]
Out[9]: [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

Ici, l'objet renvoyé est une liste qu'on appelle aussi tableau, mais qui n'est *pas* un `array` au sens de `numpy`, même si cette dernière bibliothèque fait souvent elle-même les conversions.

2 Avec la fonction `linspace` à laquelle on donne des bornes et un nombre de valeurs attendues :

```
In [10]: numpy.linspace(0,1,11)
Out[10]: array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

3 Avec la fonction `arange` qui est une adaptation de `range` pour les `array`. On note que la borne supérieure de l'intervalle est exclue comme avec `range`.

```
In [11]: numpy.arange(0,1.1,0.1)
Out[11]: array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

EN PRATIQUE Dans les versions Python 2.x
Attention, on se souvient qu'en Python 2.x, si *k* est un entier, alors *k*/10 va renvoyer le quotient dans la division euclidienne. D'où l'utilisation de *k*/10. et non *k*/10 si on veut que le code soit correctement interprété en Python 2.

9.3.2 Représentation de graphes avec `plot`

Pour représenter une solution, on va utiliser la commande **plot** de la bibliothèque **matplotlib.pyplot**. On suppose avoir chargé cette bibliothèque via **import matplotlib.pyplot as pl**, ce qui évite d'avoir à taper tout le préfixe à chaque fois. On suppose de même que **numpy** a été chargé via **import numpy as np**. L'utilisation de base de **plot** est : **plot(*x*,*y*)**, où *x* et *y* sont des tableaux d'abscisses et d'ordonnées. Par exemple, pour représenter le graphe de $t \mapsto e^t \sin t$ sur $[-\pi, \pi]$, on commence par créer le tableau des abscisses ; ensuite, on crée le tableau des ordonnées en appliquant la fonction f préalablement définie à chaque élément de x. Pour cela, on va appliquer directement f à un tableau, ce qui produit le tableau des images.

```
def f(t):
    return np.sin(5*t) * np.exp(t)

x = np.linspace(-np.pi, np.pi, 100)
y = f(x)
```

Appliquer directement une fonction à un tableau est assez crispant au début, mais on s'y habitue ! Cela dit, toutes les fonctions ne peuvent pas être appliquées à des tableaux. Par exemple, si la fonction **sin** utilisée avait été extraite de la bibliothèque **math**, on n'aurait pas pu appliquer f à x. Il y a alors deux possibilités : on utilise la fonction **map** (solution naturelle pour l'informaticien) avec **y = map(f, x)** ou bien on *vectorialise* f pour en faire une fonction applicable à un tableau :

```
fv = np.vectorize(f)
y = fv(x)
```

On peut alors représenter le graphe : **pl.plot(*x*, *y*)**. Le résultat est très décevant !

```
In [12]: pl.plot(x, y)

Out[12]: [<matplotlib.lines.Line2D object at 0xa428eac>]
```

Le graphe a effectivement été créé... mais pas affiché ! Pour le voir dans une fenêtre, on utilise **pl.show()**. Pour obtenir la figure 9.8, quelques décorations optionnelles ont été ajoutées : une grille, un titre, un nom pour les axes, ainsi que le tracé des axes, qui ne se fait pas par défaut. On peut observer le code complet au programme 8 ci-après.
Signalons enfin que la gestion de l'affichage, via **pl.show()**, est dépendante de l'environnement (Spyder, Idle...), ce qui est assez pénible. Une façon fiable d'accéder aux graphiques produits consiste à les sauvegarder (au format de son choix) via **pl.savefig('le-nom.pdf')**.

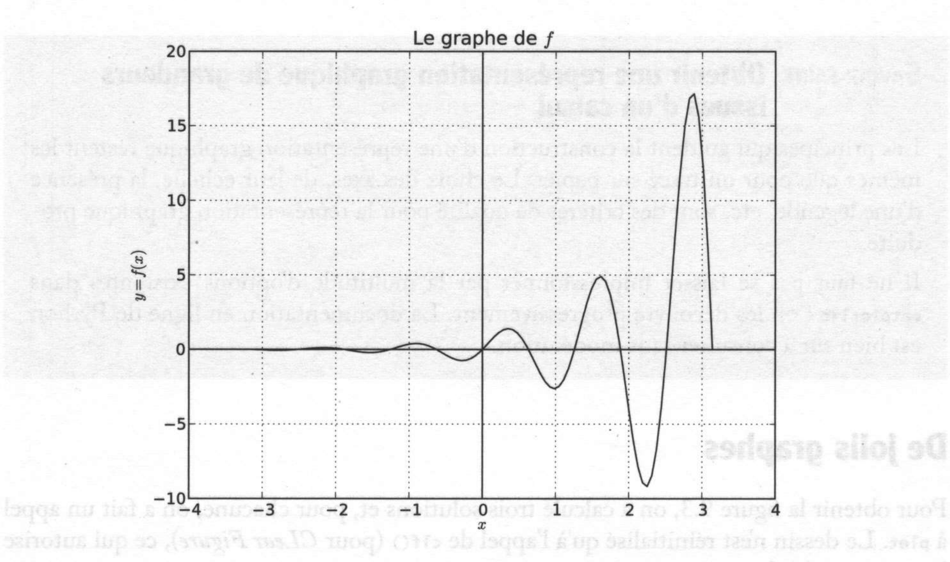


Figure 9.8
Graphe d'une fonction, avec un titre, une grille, etc.

PROGRAMME 8 Utilisation de matplotlib

```
import matplotlib.pyplot as pl
import numpy as np

def f(t):
    return np.sin(5*t) * np.exp(t)

x = np.linspace(-np.pi, np.pi, 100)
y = f(x)
pl.plot(x, y)

pl.grid()  # décoration : pour créer une grille
pl.title('Le graphe de $f$') # Le titre
pl.xlabel('$x$') # le nom de l'axe horizontal
pl.ylabel('$y=f(x)$')

pl.axhline(color='black') #linewidth=1, color='k')
pl.axvline(color='black')

pl.savefig('mon-premier-plot.pdf') # pour sauver (en pdf ou autre)
pl.show() # le show doit etre APRES la sauvegarde !
```

SAVOIR-FAIRE Obtenir une représentation graphique de grandeurs issues d'un calcul

Les principes qui guident la construction d'une représentation graphique restent les mêmes que pour un tracé sur papier. Le choix des axes, de leur échelle, la présence d'une légende, etc, sont des critères de qualité pour la représentation graphique produite.

Il ne faut pas se laisser impressionner par la multitude d'options existantes dans `matplotlib` : on les découvre progressivement. La documentation en ligne de Python est bien sûr à consulter sans modération.

9.3.3 De jolis graphes

Pour obtenir la figure 9.3, on a calculé trois solutions et, pour chacune, on a fait un appel à `plot`. Le dessin n'est réinitialisé qu'à l'appel de `clf()` (pour *CLear Figure*), ce qui autorise les tracés multiples.

```
for h in [1, 0.1, 0.01]:
    et, ey = euler(lambda t, y:y, 0, 3, 1, h)
    pl.plot(et, ey)
```

Exercice 9.5 Le code réel diffère légèrement de celui présenté ici. Tester les lignes de code précédentes, expliquer le problème et le réparer !

Pour les systèmes d'ordre 2 ou plus, on va prendre l'exemple du pendule non amorti linéarisé ($y'' = -y$). Pour obtenir la figure 9.6, on commence par définir la fonction vectorielle qui va vérifier l'équation linéaire d'ordre 1 :

```
def fpendule(t, X):  # cas linéarisé : y''=-y
    theta, thetap = X
    return array([thetap, -theta])

t, theta = euler_vectoriel(fpendule, 0, 10, array([0,1]), 10**(-2))
```

Ensuite, il y a un traitement à faire sur le résultat : `theta` est un tableau de tableaux à deux éléments. Pour extraire la première composante, on va réaliser du *slicing*, qui est une opération naturelle sur les tableaux : si on définit `x = np.array([[1,2],[3,4],[5,6]])`, on peut extraire la colonne numéro 0 :

```
In [13]: x[:, 0]
Out[13]: array([1, 3, 5])
```

mais aussi la ligne numéro 1 :

```
In [14]: x[1, :]
Out[14]: array([3, 4])
```

Comme on voulait extraire la première composante des tableaux rendus (qui représentent θ), on a écrit :

```
pl.plot(t, array(theta)[:, 0])
```

La transformation de `theta` en tableau `array` était nécessaire : le slicing ne fonctionne pas sur les tableaux de tableaux natifs de Python.

On peut, sur ce même exemple, souhaiter représenter le *portrait de phase*, c'est-à-dire l'ensemble des points des couples $(y(t), y'(t))$ pour $t \in [0, 10]$. C'est quelque chose de facile à faire, grâce à un *double-slicing*. Le programme 9 ci-après regroupe les commandes qui ont permis de réaliser la figure 9.9. On voit en particulier une option qui change l'épaisseur des traits.

Programme 9 Un portrait de phase

```
def fpendule(t, X):
    theta, thetap = X
    return array([thetap, -theta])

_,theta = euler_vectoriel(fpendule, 0, 30, array([0,1]), 0.1)
pl.plot(array(theta)[:, 0], array(theta)[:, 1])

_,theta = heun(fpendule, 0, 30, array([0,1]), 0.1)
pl.plot(array(theta)[:, 0], array(theta)[:, 1], linewidth = 4)
```

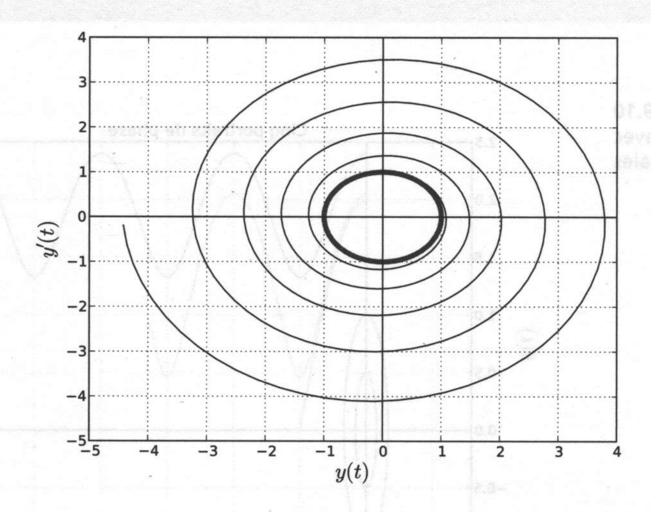


Figure 9.9
Le portrait de phase du pendule linéarisé avec un pas grossier ($h = 0.1$) : Euler vs Heun

On va terminer avec le pendule non amorti (et non linéarisé), pour lequel on présente des portraits de phase calculés avec **odeint** pour différentes valeurs de la vitesse initiale (mais toujours $\theta(0) = 0$) : le code ayant produit la figure 9.10 est donné programme 10 ci-après. Les résultats confirment l'intuition et la théorie :

- Si l'impulsion initiale est faible, le mouvement est périodique ; on retrouve l'aspect observé dans le cas linéarisé.
- Si l'impulsion initiale est forte, le mouvement est pseudo-périodique : il existe $T > 0$ tel que $\theta(t+T) = \theta(t) + 2\pi$ et $\dot{\theta}(t+T) = \dot{\theta}(t)$ pour tout t.
- Dans le cas limite (qui est fort difficile à expérimenter !), $\big(\theta(t), \dot{\theta}(t)\big)$ tend vers $(\pi, 0)$ lorsque t tend vers $+\infty$. Nous reviendrons sur ce point plus tard.

Programme 10 Portraits de phase avec odeint

```
def fpendule(X, _):
    theta, thetap = X[0], X[1]
    return array([thetap, -np.sin(theta)])

t = np.arange(0, 10, 0.01)

for theta_point_0 in [0.5, 1, 2, 2.05, 2.4]:
    r = odeint(fpendule, array([0, theta_point_0]), t)
    pl.plot(r[:, 0], r[:, 1])

# (...)
pl.xlabel(r'$\theta(t)$', fontsize=18)
# ...
```

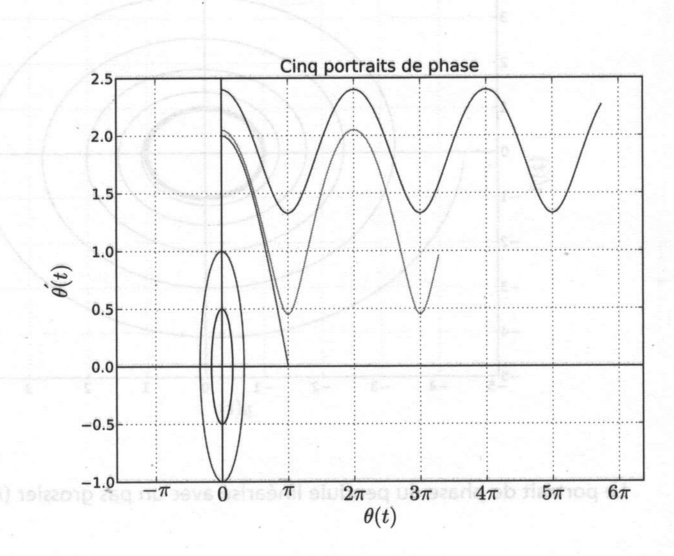


Figure 9.10
Le pendule non amorti, avec différentes conditions initiales

9.3.4 Où on observe quelques limitations

On reprend le cas limite du pendule non amorti. La figure 9.11 a été obtenue sur la plage de temps $[0, 30]$ avec quatre méthodes de résolution différentes et les pas utilisés pour les résolutions hors `odeint` valent 0.01. Aucune méthode ne renvoie une solution convergeant vers π. Lorsque $\big(\theta(t), \dot\theta(t)\big)$ est proche de $(\pi^-, 0^+)$, les approximations peuvent faire que ce couple prend une valeur de la forme $(\pi + \varepsilon_1, \varepsilon_2)$ $(\varepsilon_1, \varepsilon_2 > 0)$: le pendule bascule de l'autre côté. Cependant, $\big(\theta(t), \dot\theta(t)\big)$ peut prendre une valeur de la forme $(\pi - \varepsilon_1, -\varepsilon_2)$ et le pendule revient en arrière ! C'est ce qui se passe lors de la résolution avec la méthode de Runge-Kutta d'ordre 4.

Finalement, la résolution numérique conduit aux mêmes résultats que l'expérimentation !

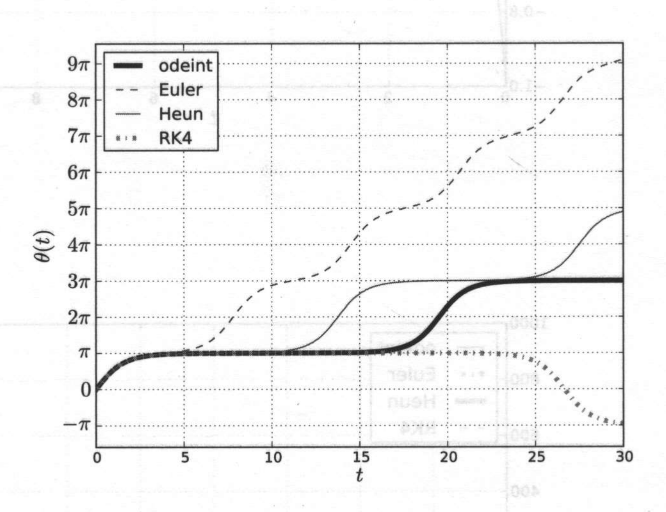


Figure 9.11
Le pendule non amorti dans le cas limite : quatre résolutions différentes

On va maintenant observer l'allure des solutions de l'équation $y'' = y' + 6y$, avec les conditions initiales $y(0) = -1$ et $y'(0) = 2$. La solution est l'application $t \mapsto -e^{-2t}$ et une première résolution numérique confirme cela, figure 9.12. Si on regarde un peu plus loin, et avec les quatre méthodes de résolution, c'est la catastrophe : elles divergent toutes (figure 9.13). Pire : meilleure est la méthode et plus rapide est la divergence !

Exercice 9.6 avec corrigé * Expliquer le phénomène !

Les solutions de l'équation différentielle (sans la condition initiale) sont des combinaisons linéaires de $t \mapsto e^{-2t}$ *et* $t \mapsto e^{3t}$. *Si une solution vérifie* $y(t_0) = \alpha$ *et* $y'(t_0) = \beta$, *alors* y *est de la forme* $t \mapsto \gamma e^{-2t}$ *si et seulement si* $\beta = -2\alpha$. *Pour notre problème, cette condition était vérifiée. Néanmoins, dès les premières étapes de la résolution, les petites approximations font que* $\beta + 2\alpha$ *sera non nul ; très petit (de l'ordre de* 2^{-52}*), mais non nul. On cherche alors à résoudre une équation différentielle dont la solution diverge, même si le facteur devant* e^{3t} *est faible.*

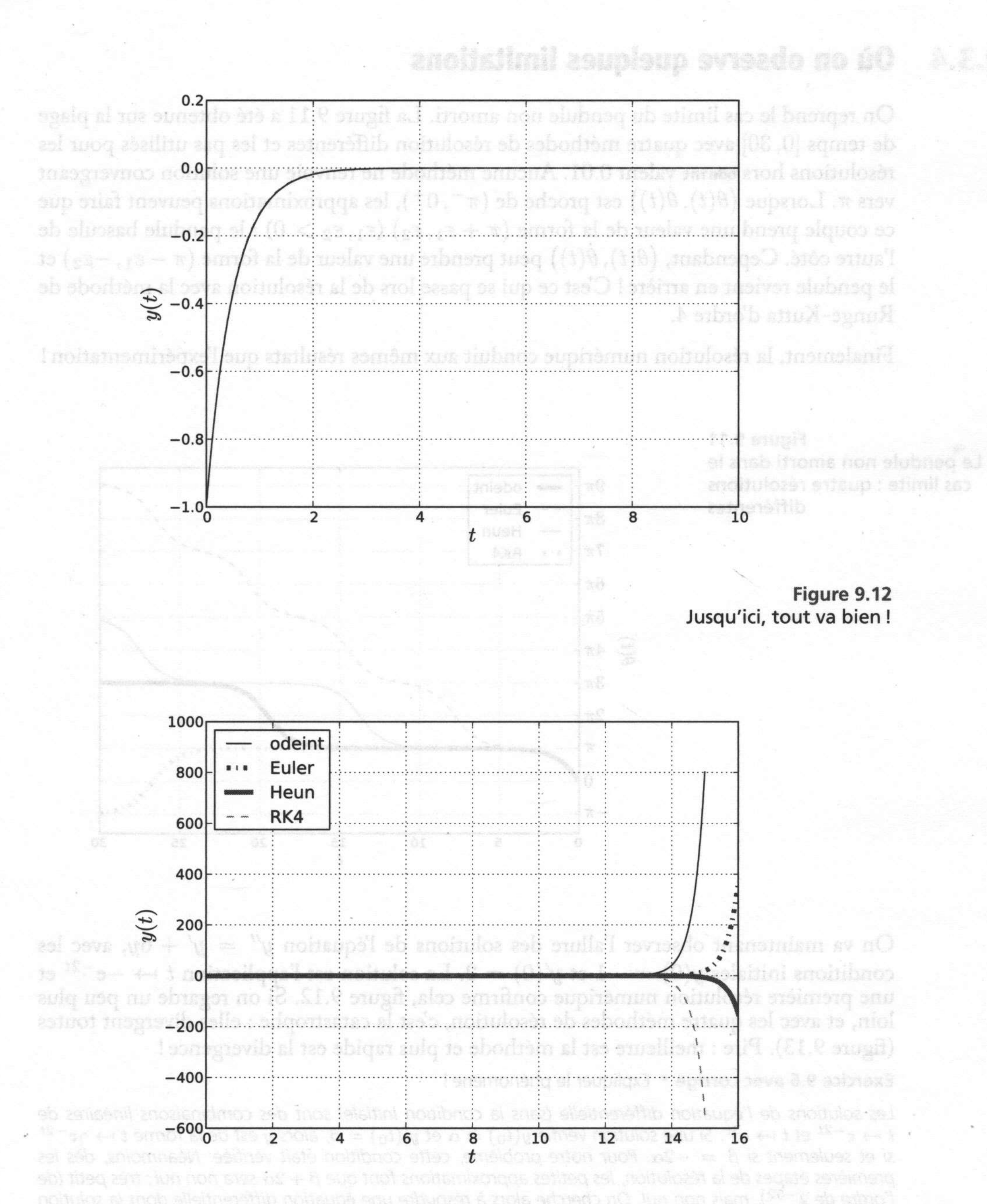


Figure 9.12
Jusqu'ici, tout va bien !

Figure 9.13
Rien à faire : les solutions numériques divergent forcément.

9.4 Exercices

Pour l'informaticien

Exercice 9.7 avec corrigé La méthode de Heun consiste à remplacer dans celle d'Euler la relation :

$$y_{k+1} = y_k + hF(t_k, y_k)$$

par :

$$y_{k+1} = y_k + \frac{h}{2}\left(F\left(t_k, y_k\right) + F\left(t_{k+1}, y_k + hF(t_k, y_k)\right)\right).$$

1 Expliquer qualitativement en quoi cela doit améliorer la précision.
2 Programmer la méthode de Heun.
3 Calculer puis représenter les solutions approchées par la méthode de Heun de $y' = y$ avec la condition initiale $y(0) = 1$.

Le calcul de y_{k+1} fait intervenir la moyenne des dérivées de y estimées en t_k et t_{k+1}. Pour une fonction convexe, la première constitue un minorant de l'accroissement de y entre t_k et t_{k+1}, et la seconde en constitue un majorant. La moyenne des deux est donc encadrée par ces deux bornes.

Pour la réalisation informatique, il n'y a qu'à écrire `y = y + h/2 * (F(t, y) + (F(t+h, y+h*F(t,y))))` *à la place de* `y = y + h * F(t, y)`. *Même avec des pas modestes, l'approximation n'est pas trop mauvaise (voir figure 9.14), et en tout cas bien meilleure qu'avec la méthode d'Euler.*

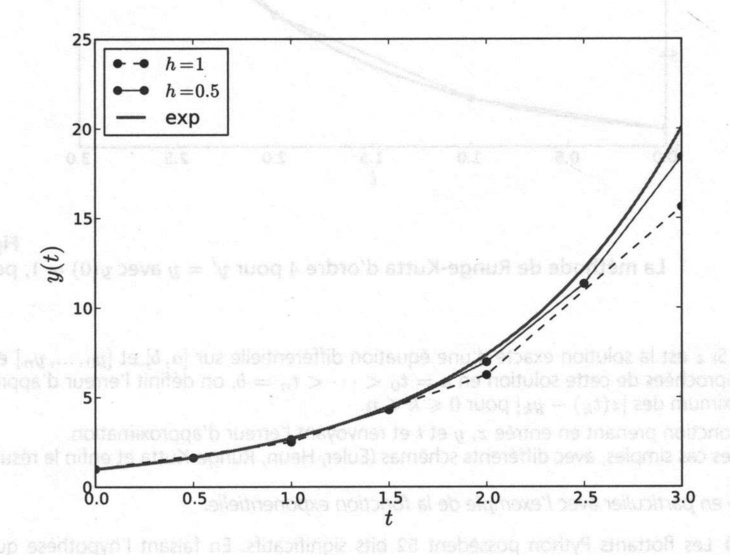


Figure 9.14
La méthode de Heun pour $y' = y$ avec $y(0) = 1$, pour $h = 1$ et $h = 0.5$

Exercice 9.8 avec corrigé Reprendre l'exercice précédent, mais avec la méthode de Runge-Kutta d'ordre 4. Cette fois, le calcul de y_{k+1} à l'aide de y_k passe par le calcul de trois valeurs intermédiaires :

$$\alpha_k = y_k + \frac{h}{2}F(t_k, y_k), \quad \beta_k = y_k + \frac{h}{2}F(t_k + h/2, \alpha_k), \quad \gamma_k = y_k + \frac{h}{2}F(t_k + h/2, \beta_k)$$

et enfin :

$$y_{k+1} = y_k + \frac{h}{6}\left(F(t_k, y_k) + 2F(t_k + h/2, \alpha_k) + 2F(t_k + h/2, \beta_k) + F(t_{k+1}, \gamma_k)\right).$$

Avec $h = 1$, l'approximation de la solution exacte est spectaculairement bonne, comme le montre la figure 9.15.

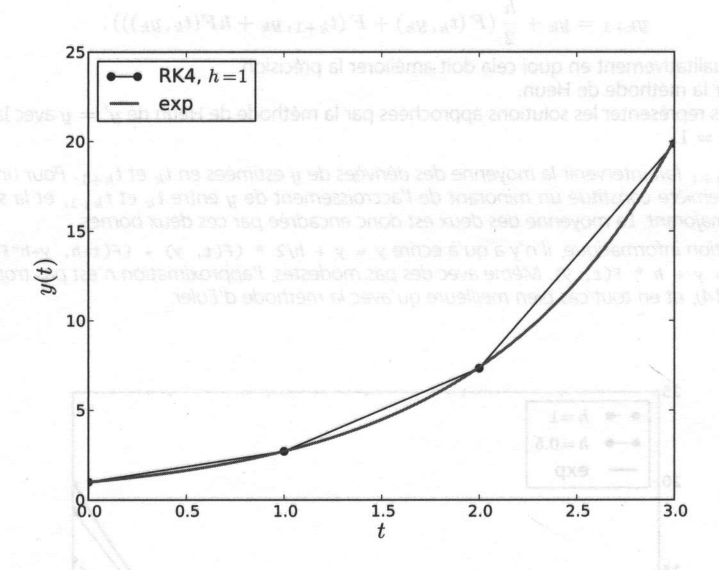


Figure 9.15
La méthode de Runge-Kutta d'ordre 4 pour $y' = y$ avec $y(0) = 1$, pour $h = 1$

Exercice 9.9 Si z est la solution exacte d'une équation différentielle sur $[a, b]$ et $[y_0, ..., y_n]$ est la suite des valeurs approchées de cette solution en $a = t_0 < \cdots < t_n = b$, on définit l'erreur d'approximation comme le maximum des $|z(t_k) - y_k|$ pour $0 \leqslant k \leqslant n$.

1 Écrire une fonction prenant en entrée z, y et t et renvoyant l'erreur d'approximation.

2 Tester sur des cas simples, avec différents schémas (Euler, Heun, Runge-Kutta et enfin le résultat fourni par `odeint`).
On vérifiera en particulier avec l'exemple de la fonction exponentielle.

Exercice 9.10 Les flottants Python possèdent 52 bits significatifs. En faisant l'hypothèse que l'erreur commise après une itération dans la méthode de résolution est de l'ordre de 2^{-52} puis qu'il n'y a plus d'erreur d'approximation, évaluer à partir de quel t on a $y(t)$ de l'ordre de 1 (début de l'explosion).
Vérifier ce calcul « brutal » avec l'équation $y'' = -2y' + 3y$ et des conditions initiales bien choisies : prévoir le moment de la divergence, puis tester.

Exercice 9.11 avec corrigé * La résolution numérique de l'équation $y'' = -2y' + 3y$ avec les conditions initiales $y(0) = 1$ et $y'(0) = -3$ produit les résultats représentés figure 9.16.
Tenter une explication. Que doit-il se passer avec l'équation translatée $y'' = -2y' + 3y - 1$ sous les conditions initiales $y(0) = \dfrac{4}{3}$ et $y'(0) = -3$? Vérifier !

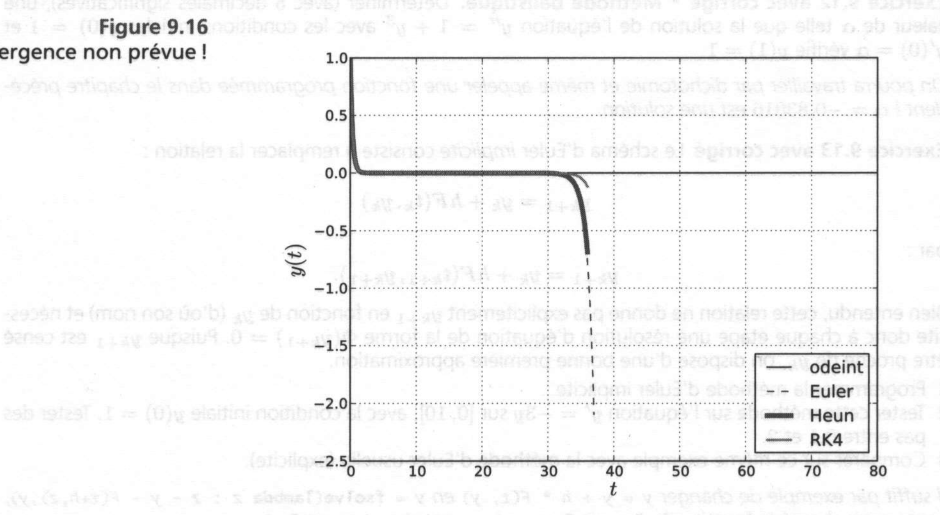


Figure 9.16
Une convergence non prévue !

La solution mathématique du problème translaté est $t \mapsto \dfrac{1}{3} + \mathrm{e}^{-3t}$. *Pourtant, le couple de flottants* $(y(t), y'(t))$ *ne pourra jamais valoir* $(1/3, 0)$... *On voit figure 9.17 que même* **`odeint`** *conduit à une solution divergente.*

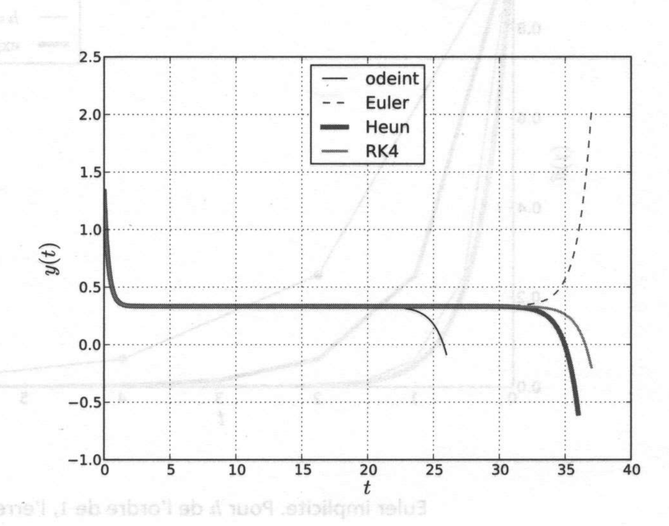


Figure 9.17
Cette fois, ça diverge comme prévu.

Exercice 9.12 avec corrigé * Méthode balistique. Déterminer (avec 5 décimales significatives), une valeur de α telle que la solution de l'équation $y'' = 1 + y^3$ avec les conditions initiales $y(0) = 1$ et $y'(0) = \alpha$ vérifie $y(1) = 1$.

On pourra travailler par dichotomie et même appeler une fonction programmée dans le chapitre précédent ! $\alpha = -0.83016$ est une solution.

Exercice 9.13 avec corrigé Le schéma d'Euler *implicite* consiste à remplacer la relation :

$$y_{k+1} = y_k + hF(t_k, y_k)$$

par :

$$y_{k+1} = y_k + hF(t_{k+1}, y_{k+1}).$$

Bien entendu, cette relation ne donne pas explicitement y_{k+1} en fonction de y_k (d'où son nom) et nécessite donc à chaque étape une résolution d'équation de la forme $\Phi(y_{k+1}) = 0$. Puisque y_{k+1} est censé être proche de y_k, on dispose d'une bonne première approximation.

1 Programmer la méthode d'Euler implicite.
2 Tester cette méthode sur l'équation $y' = -3y$ sur $[0, 10]$, avec la condition initiale $y(0) = 1$. Tester des pas entre 0.1 et 2.
3 Comparer sur ce même exemple avec la méthode d'Euler usuelle (explicite).

Il suffit par exemple de changer `y = y + h * F(t, y)` *en* `y = fsolve(lambda z : z - y - F(t+h,z),y)`, *après avoir chargé la fonction* `fsolve` *via* `from scipy.optimize import fsolve`.

On peut voir figure 9.18 que cette méthode est plus stable que la méthode explicite (figure 9.19). Elle reste cependant d'ordre 1.

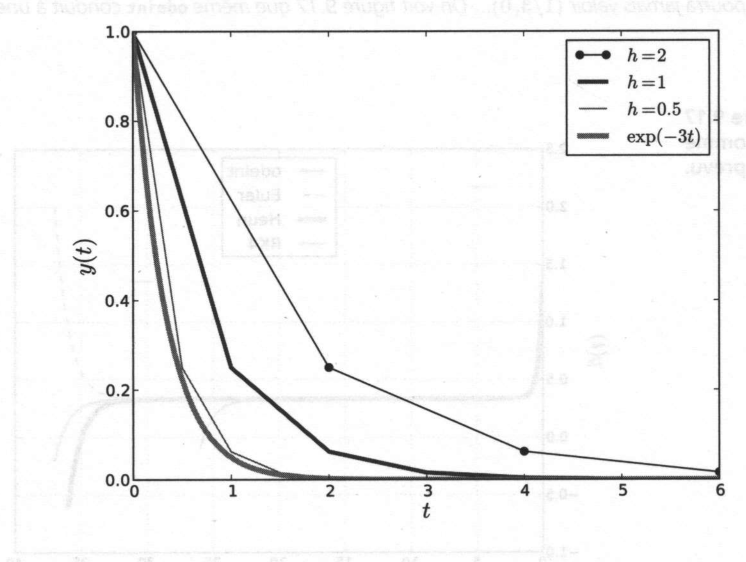


Figure 9.18
Euler implicite. Pour h de l'ordre de 1, l'erreur est assez importante.

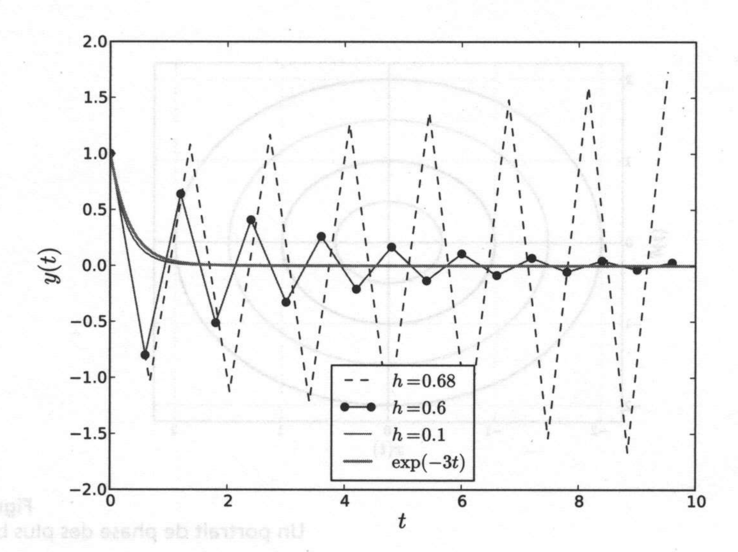


Figure 9.19
Euler explicite. Cette fois, il faut h plus petit, sans quoi il y a divergence.

Pour le mathématicien

Exercice 9.14 Montrer que pour la méthode de Heun, l'erreur de consistance dans la résolution de $y' = y$ sur $[0, 1]$ avec $y(0) = 1$ est équivalente à $\dfrac{\mathrm{e} - 1}{6n^2}$.

Exercice 9.15 avec corrigé Le portrait de phase associé à la matrice $A = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$ est représenté figure 9.20; il est constitué de trajectoires de couples (x, y) vérifiant le système différentiel autonome $Z' = AZ$, avec $Z = \begin{pmatrix} x \\ y \end{pmatrix}$.

Représenter les portraits de phase correspondant aux matrices suivantes :

$$\begin{pmatrix} -0.2 & -1 \\ 1 & -0.2 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

On pourra par exemple écrire `np.dot(A,x)` *pour faire un produit matriciel. La figure 9.21 montre ce que l'on pourra obtenir (dans le dernier cas, il faut batailler un peu pour trouver des conditions initiales satisfaisantes).*

Exercice 9.16 avec corrigé Les trois systèmes différentiels suivants correspondent à des équations de la forme $Y' = F_k(Y)$, avec $F_k(0) = 0$ et $Jac(F_k)_0 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$: ils se linéarisent tous de la même façon.

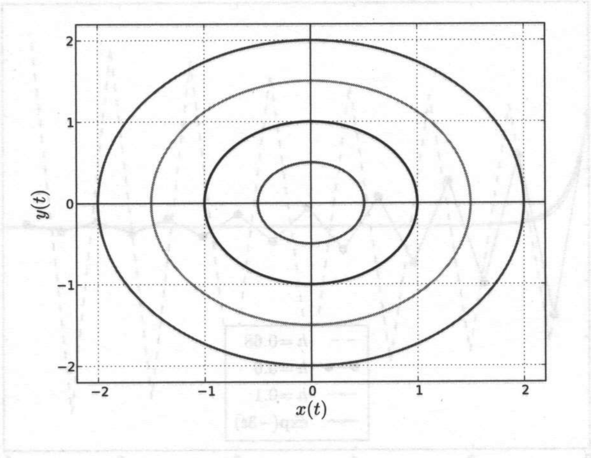


Figure 9.20
Un portrait de phase des plus basiques

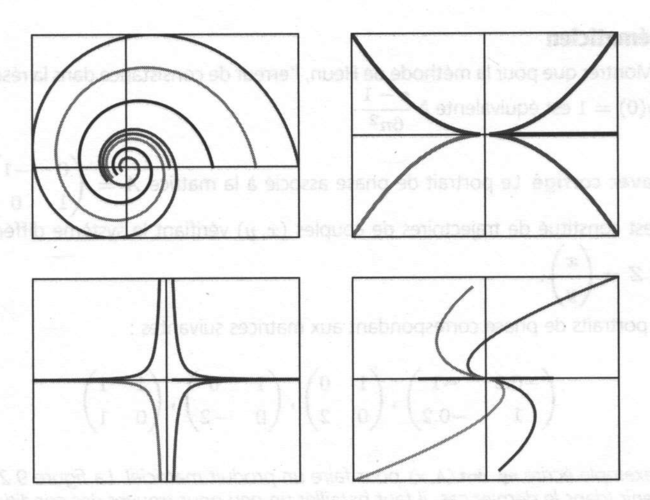


Figure 9.21
Quatre portraits de phase

$$\begin{cases} x' &= -y - x(x^2+y^2) \\ y' &= x - y(x^2+y^2) \end{cases} \quad \begin{cases} x' &= -y - \frac{x}{10}(x^2+y^2) \\ y' &= x + \frac{y}{10}(x^2+y^2) \end{cases} \quad \begin{cases} x' &= -y + \frac{x}{20}(x^2+y^2) \\ y' &= x + \frac{y}{20}(x^2+y^2) \end{cases}$$

Visualiser les courbes intégrales issues de $(1, 0)$ dans les trois cas et conclure quant au danger des linéarisations à tout va !

Accessoirement, le lecteur pourra résoudre explicitement les équations via un passage en polaire : le théorème de relèvement autorise à écrire $x(t) + iy(t) = \rho(t)e^{i\theta(t)}$ avec ρ et θ de classe $\mathcal{C}^1$ et qui vérifient des équations différentielles simples.

Les courbes sont présentées figure 9.22.

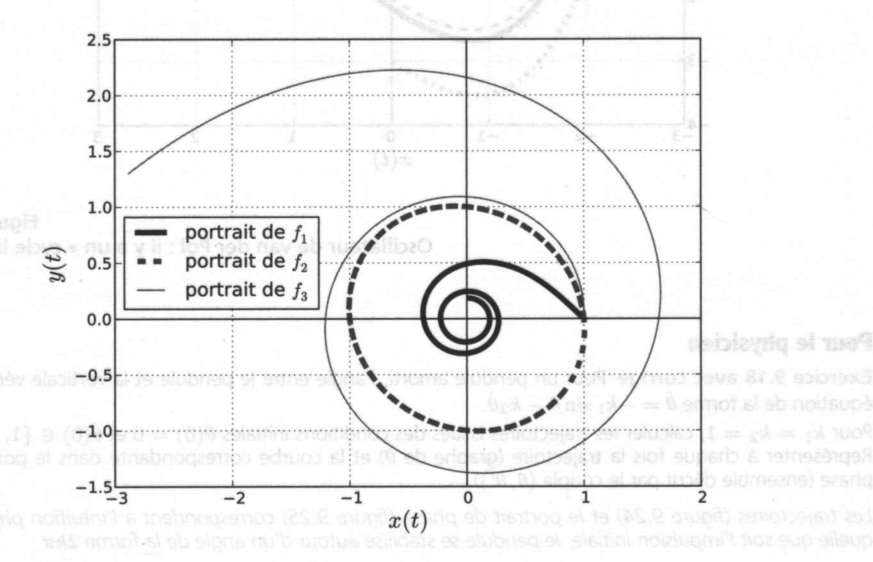


Figure 9.22
Trois comportements différents, malgré des linéarisations identiques

Exercice 9.17 avec corrigé Représenter le portrait de phase de l'équation de van der Pol :

$$x'' = \mu(1 - x^2)x' - x.$$

Pour obtenir la figure 9.23 , on a pris $\mu = 1$.

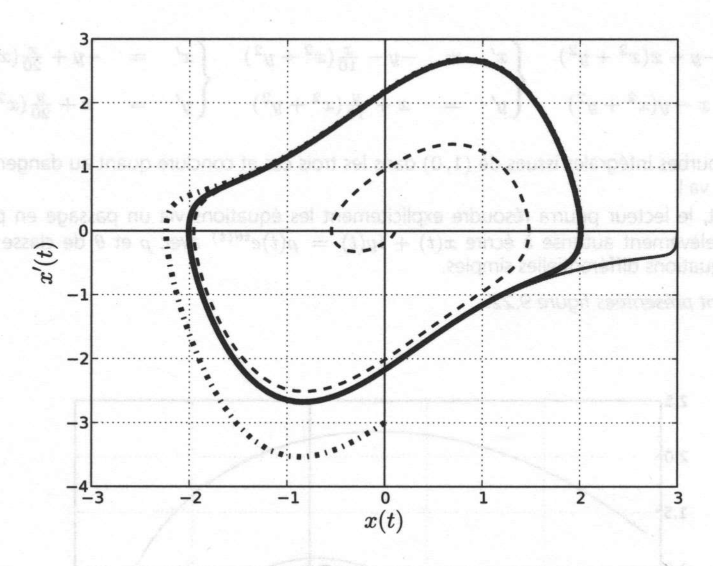


Figure 9.23
Oscillateur de van der Pol : il y a un « cycle limite ».

Pour le physicien

Exercice 9.18 avec corrigé Pour un pendule amorti, l'angle entre le pendule et la verticale vérifie une équation de la forme $\ddot{\theta} = -k_1 \sin\theta - k_2\dot{\theta}$.

Pour $k_1 = k_2 = 1$, calculer les trajectoires issues des conditions initiales $\theta(0) = 0$ et $\dot{\theta}(0) \in \{1, 2, 5, 8\}$. Représenter à chaque fois la trajectoire (graphe de θ) et la courbe correspondante dans le portrait de phase (ensemble décrit par le couple (θ, θ')).

Les trajectoires (figure 9.24) et le portrait de phase (figure 9.25) correspondent à l'intuition physique : quelle que soit l'impulsion initiale, le pendule se stabilise autour d'un angle de la forme $2k\pi$.

Exercice 9.19 avec corrigé ** Un glaçon sur un igloo. Un glaçon (supposé ponctuel) est posé en haut d'un igloo avec une vitesse initiale faible. Il est ensuite soumis à la pesanteur et à la réaction de l'igloo.

Dans un premier temps, on suppose qu'il n'y a pas de frottement ; la réaction de l'igloo est alors normale. Si on note respectivement O et M le centre de l'igloo et le point où se trouve le glaçon, l'angle entre la verticale et $\overrightarrow{OM}$ vérifie :

$$m \begin{pmatrix} r\,\ddot{\theta} \\ -r\dot{\theta}^2 \end{pmatrix} = mg \begin{pmatrix} \sin\theta \\ -\cos\theta \end{pmatrix} + \begin{pmatrix} 0 \\ N \end{pmatrix}$$

Cette équation est vérifiée tant que $N > 0$. Quand N s'annule, le glaçon perd le contact avec l'igloo.

On pourra prendre $g = 9.8$ et $m = 1$ (un beau glaçon, donc !)

1 Résoudre numériquement cette équation différentielle.

2 Vérifier « expérimentalement » que l'angle de décrochage tend vers $\arccos\dfrac{2}{3}$ lorsque la vitesse initiale tend vers 0.

Figure 9.24
Trajectoires des solutions θ pour le pendule amorti

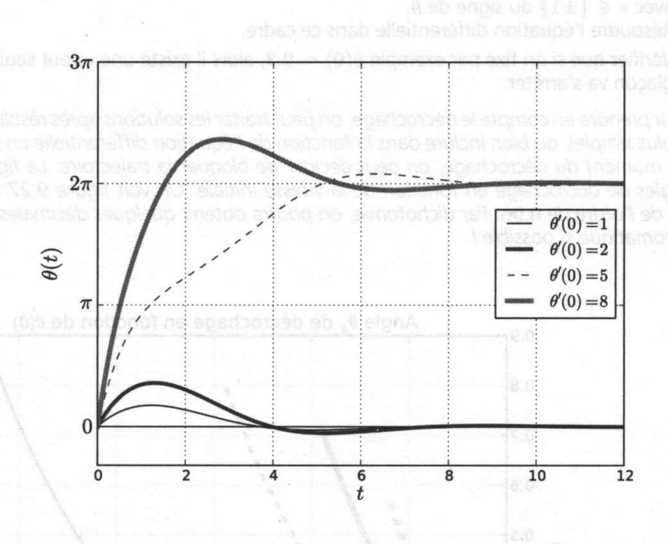
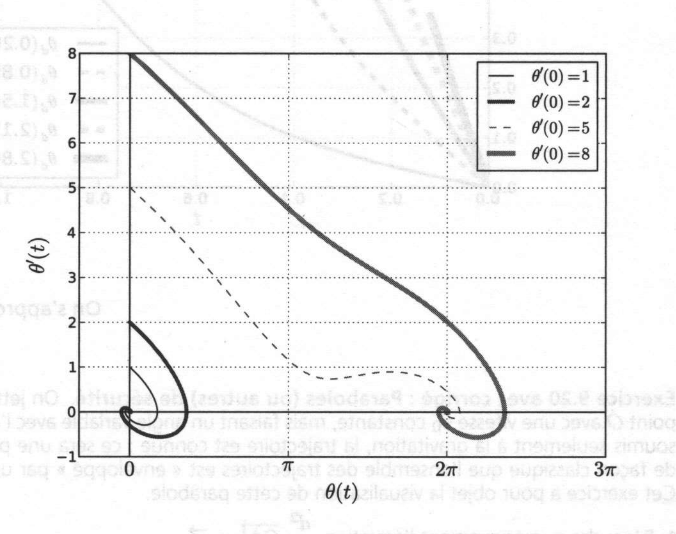


Figure 9.25
Portrait de phase pour le pendule amorti



3 On suppose maintenant que le glaçon est soumis à une force de frottement, ce qui conduit à l'équation (avant décrochage) :

$$m\begin{pmatrix} r\ddot{\theta} \\ -r\dot{\theta}^2 \end{pmatrix} = mg\begin{pmatrix} \sin\theta \\ -\cos\theta \end{pmatrix} + \begin{pmatrix} -\varepsilon kN \\ N \end{pmatrix}$$

avec $\varepsilon \in \{\pm 1\}$ du signe de $\dot{\theta}$.

4 Résoudre l'équation différentielle dans ce cadre.

5 Vérifier que si on fixe par exemple $\dot{\theta}(0) = 0.3$, alors il existe une valeur seuil de k au-delà de laquelle le glaçon va s'arrêter.

Pour prendre en compte le décrochage, on peut traiter les solutions après résolution (sans test interne, c'est le plus simple), ou bien inclure dans la fonction de l'équation différentielle un test vérifiant le décrochage. Au moment du décrochage, on peut décider de bloquer la trajectoire. La figure 9.26 montre différents angles de décrochage en fonction de la vitesse initiale. On voit figure 9.27 que le seuil critique pour k *est de l'ordre de* 0.96. *Par dichotomie, on pourra obtenir quelques décimales supplémentaires. De façon automatique si possible !*

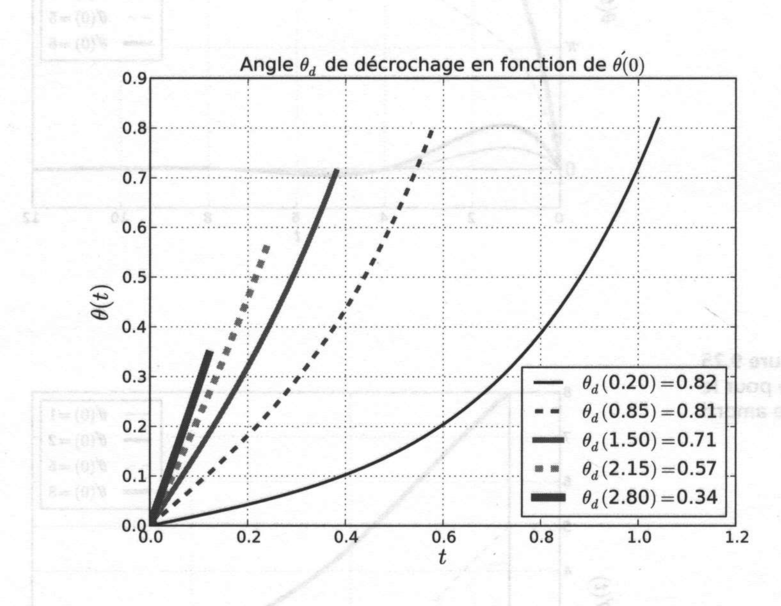


Figure 9.26
On s'approche de $\arccos(3/2) \simeq 0.84$.

Exercice 9.20 avec corrigé : Paraboles (ou autres) de sécurité. On jette un projectile M à partir du point O avec une vitesse v_0 constante, mais faisant un angle variable avec l'horizontale. Si le projectile est soumis seulement à la gravitation, la trajectoire est connue : ce sera une parabole. On démontre même de façon classique que l'ensemble des trajectoires est « enveloppé » par une parabole, dite de sécurité. Cet exercice a pour objet la visualisation de cette parabole.

1 Résoudre numériquement l'équation $\dfrac{d^2}{dt^2}\overrightarrow{OM} = \overrightarrow{g}$.

2 Représenter une vingtaine de paraboles, avec des angles variables.

3 On suppose maintenant que le projectile est soumis à une force de frottement de la forme $-k\overrightarrow{v}$. Modifier les équations et représenter à nouveau quelques trajectoires du projectile.

Ici, on pourra stopper les trajectoires dès que z *devient strictement négatif. Sans frottement, on voit la parabole se dessiner sur la figure 9.28. Sur la figure 9.29, on voit l'effet des frottements, qui raccourcissent les trajectoires.*

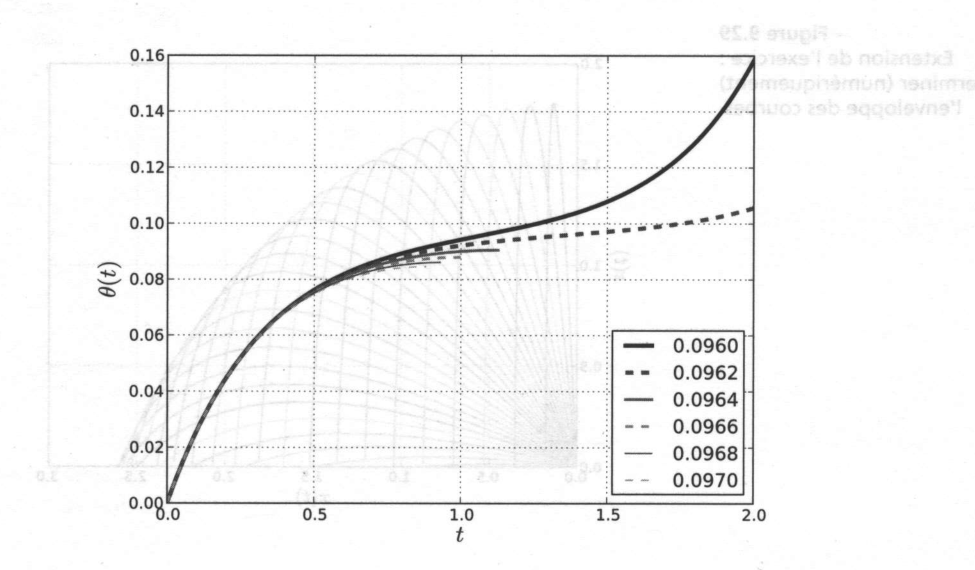


Figure 9.27
Lorsque k est trop grand, le glaçon s'arrête.

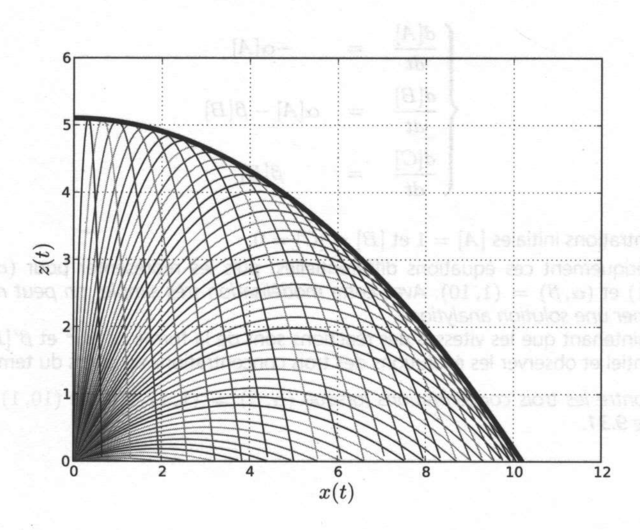
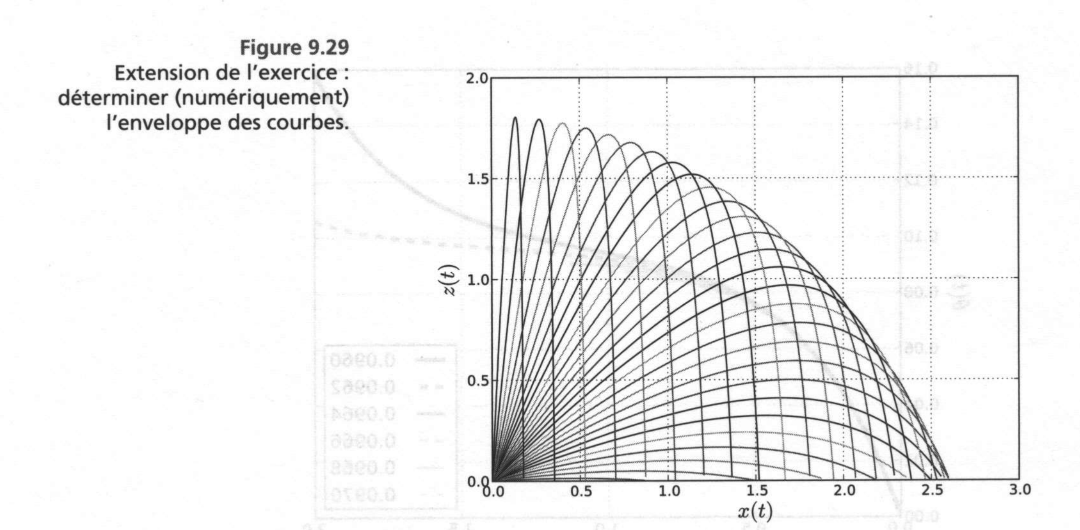


Figure 9.28
En gras, la parabole de sécurité, d'équation $z = \dfrac{v_0^2}{2g} - \dfrac{g}{2v_0^2}x^2$

Figure 9.29
Extension de l'exercice : déterminer (numériquement) l'enveloppe des courbes.



Exercice 9.21 avec corrigé : Cinétique chimique. On s'intéresse ici aux concentrations de trois produits (A, B et C) au cours du temps. Deux réactions entrent en jeu : $A \longrightarrow B$ d'une part et $B \longrightarrow C$ d'autre part. Ces réactions sont d'ordre 1 : leur vitesse est proportionnelle à la concentration du réactif. Les concentrations respectent donc des lois de la forme :

$$\begin{cases} \dfrac{d[A]}{dt} & = & -\alpha[A] \\ \dfrac{d[B]}{dt} & = & \alpha[A] - \beta[B] \\ \dfrac{d[C]}{dt} & = & \beta[B] \end{cases}$$

On part de concentrations initiales $[A] = 1$ et $[B] = [C] = 0$.

1 Résoudre numériquement ces équations différentielles, puis les représenter pour $(\alpha, \beta) = (1, 1)$, $(\alpha, \beta) = (10, 1)$ et $(\alpha, \beta) = (1, 10)$. *Avec cette modélisation très simple, on peut même sans trop de mal déterminer une solution analytique.*

2 On suppose maintenant que les vitesses des réactions sont de la forme $\alpha'[A]^2$ et $\beta'[B]^2$. Modifier le système différentiel et observer les évolutions des trois concentrations au cours du temps.

La figure 9.30 montre les trois concentrations lorsque $(\alpha, \beta) = (1, 1)$. Les cas $(10, 1)$ et $(1, 10)$ sont traités sur la figure 9.31.

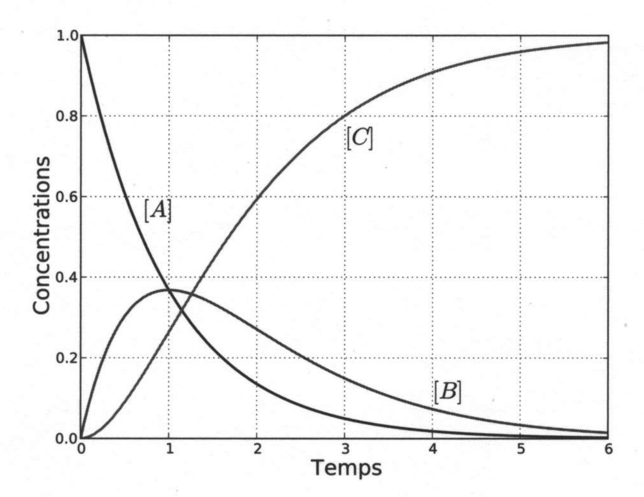


Figure 9.30
Évolution de trois concentrations, avec deux réactions de vitesse comparable

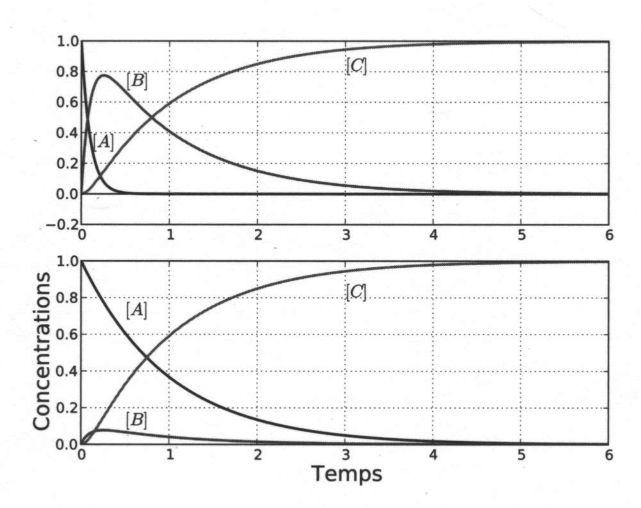


Figure 9.31
Dans ces deux cas, l'une des deux réactions est sensiblement plus rapide que l'autre.

Quatrième partie

Bases de données

Dans cette partie, nous nous intéressons à une représentation de l'information à la fois plus complexe et plus en lien avec les applications industrielles, par le biais du modèle relationnel des bases de données. Nous montrons comment exprimer, dans le langage de l'algèbre relationnelle, des requêtes de recherche d'abord simples (chapitre 10), puis faisant intervenir plusieurs relations (chapitre 11) et nous abordons la traduction de ces requêtes dans le langage SQL.

10

Algèbre relationnelle

Dans ce chapitre, nous présentons un modèle à base de relations qui sert à structurer des données non hiérarchiques. Nous définissons des opérateurs sur ces relations, de façon algébrique, et nous montrons comment les utiliser pour effectuer des requêtes dans une table.

10.1 Limites des structures de données plates pour la recherche d'informations

L'utilité de recourir à un modèle relationnel sera illustrée au fil de l'exemple réaliste suivant, où on verra que les structures plates telles que les tableaux ne conviennent pas pour représenter et surtout rechercher certains types d'informations.

- On souhaite représenter l'ensemble des élèves de CPGE d'un lycée tout en sachant qu'ils sont regroupés par classe.
- Les élèves et les classes ont des attributs propres : le nom d'un élève, le lycée où il a effectué sa terminale, la filière d'une classe (MPSI, PCSI...), le numéro de la classe au sein de l'établissement (MPSI 2...).
- On veut pouvoir rechercher facilement :
  - les élèves appartenant à une même classe ;
  - les élèves partageant un attribut commun, par exemple ceux ayant effectué leur terminale dans un lycée donné.

On prend pour exemple les trois classes préparatoires d'un lycée nommé Charles-Le-Petit :

- MPSI 1, élèves :
  - Évariste (terminale : Lycée Adams)
  - Léa (terminale : Lycée Cleese)
- MPSI 2, élèves :
  - Coralie (terminale : Lycée Adams)
  - Augustin (terminale : Lycée Chapman)
- PCSI 1, élèves :
  - Johanna (terminale : Lycée Cleese)
  - Pierre (terminale : Lycée Adams)

On peut présenter cette structure par un tableau de classes contenant chacune un tableau d'élèves :

```
lycee = [
    ("MPSI", 1, [
        ("Évariste", "Lycée Adams"),
        ("Léa", "Lycée Cleese")
        ]),
    ("MPSI", 2, [
        ("Coralie", "Lycée Adams"),
        ("Augustin", "Lycée Chapman")
        ]),
    ("PCSI", 1, [
        ("Johanna", "Lycée Cleese"),
        ("Pierre", "Lycée Adams")
        ])
    ]
```

Récupérer la liste des élèves d'une classe est très simple. Il suffit de parcourir le tableau des classes jusqu'à obtenir celle qu'on veut, puis de renvoyer son sous-tableau d'élèves :

```
def recherche_classe(filiere, numero):
    for classe in lycee:
        if classe[0] == filiere and classe[1] == numero:
            return classe[2]
    return None
```

Pour obtenir l'ensemble des élèves ayant passé leur terminale dans le lycée Adams, il faut parcourir chaque classe et filtrer les élèves concernés. Cette recherche est plus complexe que la précédente ; on constate notamment qu'il faut effectuer deux boucles `for` imbriquées : une sur les classes, puis une sur leurs élèves.

```
def recherche_term(term):
    eleves = []
    for classe in lycee:
        for eleve in classe[2]:
            if eleve[1] == term:
                eleves.append(eleve)
    return eleves
```

On note que la représentation des données est implicitement choisie pour faciliter le type de recherche prévu.

Si on choisit une autre représentation, en créant un tableau des lycées et en leur associant les élèves qui y étaient en terminale, c'est la recherche des élèves provenant du même lycée qui sera facilitée. En revanche, pour trouver l'ensemble des élèves d'une classe, il faudra parcourir tous les lycées et tous les élèves.

Dans cet exemple, un élève a des liens d'appartenance à la fois avec les classes et avec les lycées. Or, la représentation du problème sous forme de tableau contraint à privilégier un de ces liens. De plus, on veut pouvoir effectuer des recherches indépendamment de tout lien, par exemple rechercher tous les élèves portant le même nom.

10.2 Représentation dans le modèle relationnel

En faisant abstraction de tout lien d'appartenance, il est possible de représenter de manière assez similaire un élève et une classe. En effet, on pourrait considérer qu'une classe est un couple de la forme *(filière, numéro)*, par exemple :

("MPSI", 1), ("MPSI", 2), ("PCSI", 1)

et les élèves des triplets de la forme *(nom, classe, lycée-de-terminale)*, par exemple :

("Évariste", "MPSI 1", "Lycée Adams"), ("Léa", "MPSI 1", "Lycée Cleese")…

Chacune de ces représentations est appelée *relation*, ou *table* dans le vocabulaire des bases de données. Cette représentation rappelle la notion de relation binaire vue dans le cadre du cours de mathématiques.

On considère donné un ensemble fini $\mathcal{A}$, dont les éléments sont appelés des *attributs*, un ensemble $\mathbf{D}$ et une application dom de $\mathcal{A}$ dans les sous-ensembles de $\mathbf{D}$. Si $A \in \mathcal{A}$, $\mathsf{dom}(A)$ est appelé le *domaine* de A. Cette notion est à rapprocher de la notion de *types* dans les langages de programmation.

Dans le cadre de l'exemple précédent, le numéro d'une classe est un attribut dont le domaine est constitué des entiers strictement positifs ; la filière d'une classe est un attribut dont le domaine est l'ensemble des filières de CPGE.

On appelle *schéma relationnel* un n-uplet de la forme $S = (A_1, \cdots, A_n) \in \mathcal{A}^n$ où les A_i sont **distincts deux à deux**. On pourra noter $S = ((A_1, \mathsf{dom}(A_1)), \cdots, (A_n, \mathsf{dom}(A_n)))$ pour rappeler les domaines de ses attributs.

Ainsi, pour la représentation des classes, les attributs sont **filière** et **numéro**. Le schéma est le suivant :

$$((\textbf{filière}, \{\text{MPSI, PCSI}\ldots\}), (\textbf{numéro}, \mathbb{N}))$$

On notera $B \in S$ pour dire que B est un des attributs de S. Par extension, si $X = \{B_1, \cdots, B_m\}$ où les B_i sont des attributs distincts, on notera $X \subset S$ pour dire que $X \subset \{A_1, \cdots, A_n\}$ si $S = (A_1, \cdots, A_n)$. On remarque qu'il est alors possible de voir X comme un sous-n-uplet de S. Ainsi, si :

$$S = (\textbf{id, nom, prénom, adresse, code-postal, ville})$$

on pourra noter $X = \{$ **prénom, ville, nom** $\} \subset S$ sous la forme :

$$X = (\textbf{nom, prénom, ville}) \subset S$$

où les attributs sont énumérés dans le même ordre que dans S.

On appelle *relation*, ou *table*, associée à un schéma relationnel $(A_1, \cdots, A_n)$, un ensemble fini de n-uplets de $\mathsf{dom}(A_1) \times \cdots \times \mathsf{dom}(A_n)$. On note $R(S)$ la relation R pour signifier qu'elle est associée au schéma relationnel S. Les éléments de R sont appelés *valeurs* ou *enregistrements* de la relation ; leur nombre est appelé *cardinal* de la relation et est noté $\#R$.

Dans la mesure où le nombre de n-uplets est fini, on peut représenter la relation sous la forme d'un tableau :

classe

Filière	Numéro
MPSI	1
MPSI	2
PCSI	1

On va considérer un autre exemple : on souhaite réaliser la relation associée aux livres dans une bibliothèque. On pourra par exemple prendre le schéma relationnel suivant :

$$((\text{Titre}, \text{Texte}), (\text{Auteur}, \text{Texte}), (\text{Année-de-publication}, \mathbb{Z}))$$

où Texte désigne l'ensemble des phrases que l'on peut entrer à l'aide d'un clavier.

Ce schéma pourra être représenté par le diagramme suivant :



Voici une relation correspondant à l'inventaire d'une bibliothèque :

inventaire

Titre	Auteur	Année-de-publication
Éléments	Euclide	-300
Traité du calcul différentiel	Euler	1755

Si $e \in R(S)$ et $A \in S$, on note $e.A$ la composante du n-uplet e associée à l'attribut A. Plus généralement, si $X = (A_1, \cdots, A_n) \subset S$, on note $e(X) = (e.A_1, \cdots, e.A_n)$.

Dans la mesure où R est un ensemble, deux valeurs distinctes diffèrent forcément au moins sur un attribut. On peut noter cela formellement :

$$\forall e, e' \in R(S), \text{si } e \neq e' \text{ alors } \exists\, A \in S, e.A \neq e'.A$$

Il n'est donc pas possible dans l'exemple précédent d'indiquer que la bibliothèque possède deux exemplaires d'un même livre. On peut cependant étendre le schéma avec un attribut **nombre-d'exemplaires** de domaine $\mathbb{N}$ pour résoudre ce problème.

Exercice 10.1 Proposer un domaine approprié pour des attributs représentant :
- une adresse e-mail ;
- une nationalité ;
- un âge.

Exercice 10.2 Proposer des schémas relationnels représentant respectivement :
- l'inventaire d'un supermarché ;
- le parc informatique d'une entreprise.

Dans chaque cas, donner un exemple significatif de relation associée au schéma proposé.

10.3 Opérateurs sur le modèle relationnel

10.3.1 Description des recherches

Une fois les données représentées sous forme relationnelle, il faut pouvoir les exploiter pour y rechercher des informations précises. Pour cela, on doit tout d'abord être capable d'exprimer ces recherches sous une forme précise. On va donner ici un aperçu de ce qui sera formellement défini plus loin : le langage de requêtes SQL.

On considère une relation *eleve* et l'expression suivante :

```
SELECT nom FROM eleve WHERE prenom = 'Serge' OR prenom = 'Muriel';
```

Elle correspond à une description précise de la recherche : *obtenir les noms de famille de tous les élèves se prénommant Serge ou Muriel.*

Au lieu d'essayer de construire tout de suite un algorithme permettant de répondre à cette recherche, on va essayer de découper celle-ci en opérations élémentaires réutilisables.

Pour répondre à cette recherche, on doit être capable de :

- sélectionner dans la relation *eleve* les valeurs dont l'attribut **prénom** correspond à "Serge" ou à "Muriel" ;
- regrouper ces deux ensembles de valeurs ;
- extraire de chacune des valeurs, l'attribut **nom**.

On pourrait donc décomposer cette recherche à l'aide de trois opérations génériques agissant sur des relations :

- une *sélection* permettant d'extraire une sous-relation dont les valeurs vérifient un critère donné ;
- une *union* pour regrouper les valeurs de deux relations, sous réserve que les deux relations aient le même schéma ;
- une *projection* pour ne conserver d'une relation que la valeur de certains attributs.

On appelle de telles opérations des *opérateurs relationnels*. On va présenter dans cette partie les opérateurs les plus simples ; d'autres opérateurs essentiels, mais plus complexes, feront l'objet du chapitre suivant.

Il est possible de traduire automatiquement des recherches exprimées dans un langage comme SQL à l'aide d'une succession d'opérateurs relationnels. En général, cette traduction n'est pas unique et il existe de nombreuses possibilités. Certaines de ces traductions

peuvent être coûteuses en mémoire ou en temps de calcul, alors que d'autres seront très efficaces. Déterminer une bonne traduction s'appelle *optimiser la recherche*.

On recourt à deux notations utilisées en mathématiques pour définir des ensembles. Si E est un ensemble et f une application définie sur E, on note $\{ f(x) \mid x \in E \}$ l'ensemble des images des éléments de E par f. Si E est un ensemble et P une propriété pouvant être décidée sur les éléments de E, on note $\{ x \in E \mid P(x) \}$ l'ensemble des éléments de E qui vérifient P. Par exemple, l'ensemble des entiers pairs est $\{ 2p \mid p \in \mathbb{N} \} = \{ n \in \mathbb{N} \mid \exists p \in \mathbb{N}, n = 2p \}$.

POUR ALLER PLUS LOIN Calcul relationnel

À l'aide de ces notations, on peut étendre le langage de la logique pour parler des attributs et ainsi exprimer formellement les informations que l'on recherche dans une relation.

Par exemple, on souhaite extraire tous les livres publiés par Terry Pratchett au XXIe siècle dans l'inventaire d'une bibliothèque comme celui présenté plus haut. Cette requête s'exprime sous la forme de l'ensemble :

$$\{ l \in \textit{inventaire} \mid l.\textbf{Auteur} = \text{"Pratchett" et } l.\textbf{Année-de-publication} \geqslant 2000 \}$$

L'exemple précédent sur la relation *eleve* correspondrait à l'ensemble suivant :

$$\{ e.\textbf{nom} \in \textit{eleve} \mid e.\textbf{prenom} = \text{"Serge" ou } e.\textbf{prenom} = \text{"Muriel"} \}$$

Ce langage logique s'appelle le *calcul relationnel*.

10.3.2 Opérateurs ensemblistes usuels

On a vu qu'une relation est un ensemble fini de n-uplets de $D_1 \times \cdots \times D_n$ où les D_i sont les domaines de ses attributs.

Si deux relations ont le même schéma, alors il est possible de leur appliquer des opérateurs ensemblistes. Parmi ces opérateurs, on utilisera en particulier l'union ($\cup$), l'intersection ($\cap$) et la différence ($-$).

On considère, tout au long de cette section, deux relations $livre_1$ et $livre_2$ sur lesquelles on peut appliquer les opérateurs ensemblistes.

$livre_1$

Titre	Auteur	Tome
Madame Bovary	Gustave Flaubert	1
Le comte de Monte-Cristo	Alexandre Dumas	1
Le comte de Monte-Cristo	Alexandre Dumas	2

livre$_2$

Titre	Auteur	Tome
Madame Bovary	Gustave Flaubert	1
Le père Goriot	Honoré de Balzac	1

Union

L'*union* de deux relations $R_1(S)$ et $R_2(S)$ est l'ensemble des valeurs comprises dans R_1 ou dans R_2. On la note $R_1 \cup R_2$ de schéma S.

Voici le résultat de l'union des deux relations *livre*$_1$ et *livre*$_2$:

livre$_1 \cup$ *livre*$_2$

Titre	Auteur	Tome
Madame Bovary	Gustave Flaubert	1
Le comte de Monte-Cristo	Alexandre Dumas	1
Le comte de Monte-Cristo	Alexandre Dumas	2
Madame Bovary	Gustave Flaubert	1
Le père Goriot	Honoré de Balzac	1

Intersection

L'*intersection* de deux relations $R_1(S)$ et $R_2(S)$ est l'ensemble des valeurs comprises dans R_1 et dans R_2. On la note $R_1 \cap R_2$ de schéma S.

Voici le résultat de l'intersection des deux relations *livre*$_1$ et *livre*$_2$:

livre$_1 \cap$ *livre*$_2$

Titre	Auteur	Tome
Madame Bovary	Gustave Flaubert	1

Différence

La *différence* entre deux relations $R_1(S)$ et $R_2(S)$ est l'ensemble des valeurs comprises dans R_1 mais pas dans R_2. On la note $R_1 - R_2$ de schéma S.

Voici le résultat de la différence entre les deux relations *livre*$_1$ et *livre*$_2$:

livre$_1 -$ *livre*$_2$

Titre	Auteur	Tome
Le comte de Monte-Cristo	Alexandre Dumas	1
Le comte de Monte-Cristo	Alexandre Dumas	2

POUR ALLER PLUS LOIN Opérations ensemblistes sur des schémas compatibles

Ici, on a uniquement considéré des relations dont les schémas sont identiques.

En réalité, on peut effectuer ces mêmes opérations ensemblistes dès que les schémas des relations comportent le même nombre d'attributs et que les *domaines* de ces derniers sont identiques.

On parle dans ce cas de schémas *compatibles*. Il est à noter que lorsqu'on effectue une opération ensembliste sur deux schémas compatibles, il faut choisir un schéma, et en particulier des noms d'attributs, pour la nouvelle relation construite. On prend généralement la convention d'utiliser les noms d'attributs issus de la première relation.

Une telle opération reste délicate car deux attributs (par exemple **Âge** et **Prix**) peuvent avoir le même domaine $\mathbb{N}$ sans avoir la même signification.

10.3.3 Projection

Soit $R(S)$ une relation de schéma S et $X \subset S$. On appelle *projection* de R selon X la relation :

$$\pi_X(R) = \{ e(X) \mid e \in R \}$$

Le schéma de $\pi_X(R)$ est donc X.

On considère par exemple la relation :

élève

Nom	Prénom	Classe
Meyer	Zoé	MPSI 1
Michel	Florent	MPSI 1
Benoit	Marie	PCSI 2
Michel	Zoé	PCSI 1

Sa projection sur les attributs (**Nom**, **Classe**) donne :

$\pi_{Nom,Classe}(\textit{élève})$

Nom	Classe
Meyer	MPSI 1
Michel	MPSI 1
Benoit	PCSI 2
Michel	PCSI 1

Une projection ne contient pas forcément autant de valeurs que la relation de départ. En effet, plusieurs valeurs peuvent être fusionnées. Par exemple, la projection $\pi_{\text{Classe}}(\textit{élève})$ ne contient que trois valeurs, une par classe. L'exercice suivant précise cette remarque.

Exercice 10.3 Soit $R(S)$ une relation et $X \subset S$.
Montrer que $\#\pi_X(R) \leqslant \#R$.

10.3.4 Sélection

Sélection simple

Si $R(S)$ est une relation de schéma S, $A \in S$ et $a \in \text{dom}(A)$, on appelle *sélection* de R selon $A = a$ la relation obtenue en sélectionnant dans R uniquement les valeurs e telles que $e.A = a$. On la note $\sigma_{A=a}(R)$. On a donc :

$$\sigma_{A=a}(R) = \{ \, e \in R \mid e.A = a \, \}$$

Si le domaine de l'attribut le permet, on peut étendre la notion de sélection à d'autres comparaisons que l'égalité. Par exemple, $\sigma_{A \geqslant 1}(R)$ sélectionnera les valeurs e de R telles que $e.A \geqslant 1$.

Ainsi, si on considère la relation $livre_1$ (p. 261), pour sélectionner tous les ouvrages écrits par Alexandre Dumas, on pourra effectuer la sélection :

$\sigma_{\textit{Auteur="Alexandre Dumas"}}(livre_1)$

Titre	Auteur	Tome
Le comte de Monte-Cristo	Alexandre Dumas	1
Le comte de Monte-Cristo	Alexandre Dumas	2

Comparaison entre plusieurs attributs

Pour obtenir des requêtes plus expressives, on peut effectuer les comparaisons sur des opérations entre plusieurs attributs. Par exemple, dans la relation :

stock

Caisse	Fruits	Pommes	Bananes
1	10	3	7
2	7	3	3
3	8	7	1

on peut vérifier l'intégrité de la relation (c'est-à-dire que le nombre de fruits correspond bien à la somme des pommes et des bananes) grâce à l'opération suivante, qui indique les caisses pour lesquelles le compte est erroné :

$\sigma_{\textit{Fruits} \neq \textit{Pommes}+\textit{Bananes}}(stock)$

Caisse	Fruits	Pommes	Bananes
2	7	3	3

Sélection composée

Grâce aux opérateurs ensemblistes, il est possible d'exprimer des conditions complexes. On va supposer que l'on a réussi à sélectionner les relations R_1 et R_2 composées des valeurs de R satisfaisant respectivement les conditions C_1 et C_2.

- Pour obtenir les valeurs vérifiant $C = C_1$ **ET** C_2, on calcule $R_1 \cap R_2$.
- Pour obtenir les valeurs vérifiant $C = C_1$ **OU** C_2, on calcule $R_1 \cup R_2$.
- Pour obtenir les valeurs ne vérifiant pas C_1, c'est-à-dire vérifiant $C =$ **NON** C_1, on calcule $R - R_1$.

On note alors $\sigma_C(R)$ la relation obtenue en effectuant cette décomposition d'une condition complexe à l'aide des opérateurs ensemblistes et des sélections simples.

En reprenant la relation *élève* vue précédemment, on peut réaliser la sélection complexe satisfaisant la condition :

« Soit le nom de l'élève est Michel et son prénom n'est pas Zoé, soit il est en PCSI 1. »

Elle se traduit par la relation suivante :

$$R = [\sigma_{\text{Nom="Michel"}}(\textit{élève}) - \sigma_{\text{Prénom="Zoé"}}(\sigma_{\text{Nom="Michel"}}(\textit{élève}))] \cup \sigma_{\text{Classe="PCSI1"}}(\textit{élève})$$

R

Nom	Prénom	Classe
Michel	Florent	MPSI 1
Michel	Zoé	PCSI 1

Il est à noter que, dans ce cas, on peut simplifier l'opération car :

$$\sigma_{\text{Prénom="Zoé"}}(\sigma_{\text{Nom="Michel"}}(\textit{élève})) \subset \sigma_{\text{Prénom="Zoé"}}(\textit{élève})$$

et donc :

$$\sigma_{\text{Nom="Michel"}}(\textit{élève}) - \sigma_{\text{Prénom="Zoé"}}(\sigma_{\text{Nom="Michel"}}(\textit{élève})) = \sigma_{\text{Nom="Michel"}}(\textit{élève}) - \sigma_{\text{Prénom="Zoé"}}(\textit{élève})$$

POUR ALLER PLUS LOIN Simplification des sélections composées

Plus généralement, si $R_1 \subset R_2$, alors $\sigma_C(R_1) \subset \sigma_C(R_2)$.

La simplification ci-avant se généralise également :

$$\sigma_{C_2}(R) - \sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(R) - \sigma_{C_1}(R)$$

En effet, si une valeur est dans $\sigma_{C_2}(R) - \sigma_{C_1}(\sigma_{C_2}(R))$, alors elle vérifie C_2. Comme elle n'est pas dans $\sigma_{C_1}(\sigma_{C_2}(R))$, elle ne vérifie pas C_1. Elle est donc bien dans $\sigma_{C_2}(R) - \sigma_{C_1}(R)$. L'autre inclusion est immédiate.

10.3.5 Renommage

Il est possible, souvent pour lever une ambiguïté, de renommer un attribut d'une relation à l'aide d'un opérateur dit de renommage. La relation obtenue est alors identique à la relation de départ, mis à part le schéma qui a été changé pour présenter le nouveau nom.
Soit $S = (A_1, \cdots, A_n)$ un schéma, $i \in [\![1; n]\!]$ et B un attribut tel que $\mathsf{dom}(B) = \mathsf{dom}(A_i)$. On note :

$$\rho_{A_i \leftarrow B}(S) = (A_1, \cdots, A_{i-1}, B, A_{i+1}, \cdots, A_n)$$

le schéma déduit de S en *renommant* A_i en B.

Par extension, on note :

$$\rho_{A_1,\ldots,A_n \leftarrow B_1,\ldots,B_n}(S) = \rho_{A_n \leftarrow B_n}(\rho_{A_{n-1} \leftarrow B_{n-1}}(\ldots \rho_{A_1 \leftarrow B_1}(S)\ldots))$$

Dans les valeurs d'une relation, les attributs n'apparaissent que par l'intermédiaire de leur domaine. Il est donc possible de substituer au schéma de R, un schéma obtenu par renommage.

On notera $\rho_{A_1,\ldots,A_n \leftarrow B_1,\ldots,B_n}(R(S)) = R(\rho_{A_1,\ldots,A_n \leftarrow B_1,\ldots,B_n}(S))$. Ainsi, on peut considérer l'opération de renommage comme étant définie directement sur les relations.

Dans la relation *élève* vue précédemment, en renommant **Nom** en **LastName** et **Prénom** en **FirstName**, on obtient :

$\rho_{\textit{Nom, Prénom} \leftarrow \textit{LastName, FirstName}}(\textit{élève})$

LastName	FirstName	Classe
Meyer	Zoé	MPSI 1
Michel	Florent	MPSI 1
Benoit	Marie	PCSI 2
Michel	Zoé	PCSI 1

On remarque qu'effectivement, les valeurs de la relation ne sont pas affectées par ce renommage.

10.3.6 Algèbre relationnelle

Les opérations vues précédemment peuvent se combiner entre elles pour engendrer des opérations plus complexes. L'ensemble des opérations obtenues par composition s'appelle l'*algèbre relationnelle*.

Savoir-faire Traduire dans le langage de l'algèbre relationnelle des requêtes simples écrites en langage courant

Pour traduire une requête simple, il faut bien entendu identifier sur quelle relation on travaille. Il faut ensuite distinguer, parmi les informations que contient cette relation, celles qui jouent un rôle pour la requête. On peut procéder dans l'ordre suivant :

1 Traduire les critères présents dans la requête sous forme de sélections.
2 Si nécessaire, utiliser des opérations ensemblistes pour combiner les sélections entre elles.
3 Utiliser des projections pour ne conserver que les informations utiles.

Cet ordre convient pour la plupart des requêtes courantes, mais il n'est pas forcément le plus approprié pour des raisons d'efficacité : si un attribut ne joue aucun rôle dans la requête, il est plus judicieux de l'éliminer par projection dès le début pour réduire le volume de données à traiter. De manière plus générale, le choix de l'ordre le plus efficace pour traiter les différentes opérations composant une requête s'appelle l'*optimisation* des requêtes.

Exercice 10.4 avec corrigé On considère les relations :

classe

id	Filière	Numéro	Professeur
1	MPSI	1	Euclide
2	MPSI	2	Turing
3	PCSI	1	Horner
4	PCSI	2	Euler

élève

Nom	Prénom	Classe	Note
Meyer	Romain	1	9,25
Martin	Paul	2	7,75
Robert	Marie	4	12,0
Michel	Lucile	2	11,5
Bernard	Sylvie	1	17,5
Martin	Romain	3	14,0
Meyer	Pierre	1	10,0
Dubois	Camille	3	11,5

Traduire les requêtes suivantes en opérations de l'algèbre relationnelle.

1 Obtenir la liste des filières proposées dans ce lycée.
2 Obtenir toutes les informations concernant les classes de PCSI.
3 Obtenir les prénoms des élèves des classes 1 et 3.
4 Obtenir les noms et les notes des élèves ayant eu une note inférieure à 10.

1 Il suffit d'effectuer une projection sur l'attribut correspondant. Les doublons sont fusionnés :

$$\pi_{\textit{Filière}}(\text{classe})$$

2 Il suffit de sélectionner les classes de PCSI :

$$\sigma_{\textit{Filière}=PCSI}(\text{classe})$$

3 On sélectionne les élèves selon chacune des classes, puis on ne garde que les prénoms :

$$\pi_{\textit{Prénom}}(\sigma_{\textit{Classe}=1}(\text{élève}) \cup \sigma_{\textit{Classe}=3}(\text{élève}))$$

4 On peut commencer par ne garder que les noms et les notes, puis sélectionner selon cette note :

$$\sigma_{\textit{Note}<10}(\pi_{\textit{Nom, Note}}(\text{élève}))$$

10.4 Utilisation d'un gestionnaire de bases de données relationnelles

Les logiciels de gestion de bases de données relationnelles servent à concevoir informatiquement des relations suivant un schéma relationnel, à définir des valeurs et à interroger la base pour rechercher des valeurs particulières.

Une *base de données* est constituée d'un ensemble de relations, ainsi que de leurs schémas relationnels. Elle est stockée dans un fichier ou un jeu de fichiers. Un gestionnaire de bases de données va gérer cette base et permettre à des utilisateurs d'y accéder ou de la modifier. Le gestionnaire joue donc un rôle d'intermédiaire entre la vision idéale, issue du modèle relationnel, et la réalité concrète de la base.

Dans un gestionnaire de bases de données, il est possible d'effectuer les opérations suivantes :

- créer une relation à l'aide d'un schéma – *on parle alors de table plutôt que de relation* ;
- ajouter des valeurs dans une relation ;
- modifier les valeurs d'une relation ;
- modifier le schéma d'une table – *cette opération est complexe car si on ajoute des attributs, il faut leur donner une valeur par défaut.*

Les gestionnaires de bases de données (SQLite, MySQL, PostgreSQL...) ne fournissent pour la plupart que des outils rudimentaires pour interagir avec eux (par exemple des applications console dans lesquelles on rentre de manière textuelle des commandes).

Pour faciliter les manipulations, on peut utiliser une interface graphique. Il existe de nombreux logiciels de ce type ; dans les travaux pratiques proposés dans l'annexe A, on a choisi de présenter :

- MySQL comme gestionnaire de bases de données, qui est un logiciel libre très répandu ;
- phpMyAdmin comme interface graphique, qui est à la fois très puissante et très visuelle.

10.4.1 Description du langage SQL

Conjointement à la définition des bases de données relationnelles, un langage spécifique permettant à l'utilisateur d'effectuer des requêtes sur une base a été développé.

Ce langage est nommé SQL, pour *Structured Query Language*. Il existe de nombreux gestionnaires de bases de données, différents par bien des aspects, mais tous comprennent les requêtes écrites dans ce langage.

Les requêtes visent aussi bien à éditer la base qu'à effectuer des recherches. On ne parlera ici que des requêtes de recherche.

Le langage SQL est conçu pour exprimer la plupart des opérateurs de l'algèbre relationnelle. On va indiquer ici la syntaxe de ce langage pour faire des requêtes de recherche en présentant la traduction des opérations vues précédemment. Toutes les requêtes de recherche commencent par SELECT et se terminent par un point-virgule.

10.4.2 Projection

Pour effectuer la projection $\pi_{A_1,\cdots,A_n}(R)$, on évalue la requête :

```
SELECT A1, ..., An FROM R;
```

Il est possible d'obtenir l'intégralité de la relation grâce à l'utilisation du joker * à la place des attributs :

```
SELECT * FROM R;
```

10.4.3 Sélection

En fait, la commande SELECT admet un paramètre supplémentaire, WHERE, qui impose une condition. La sélection s'écrit donc :

```
SELECT * FROM R WHERE A=a;
```

La commande SELECT de SQL réalise donc simultanément une sélection et une projection. Cela rend les requêtes plus concises car, comme on l'a remarqué dans la section précédente, ces deux opérations vont souvent de pair.

Pour les sélections composées, il est possible d'utiliser directement les opérateurs booléens dans la condition. Par exemple, la condition :

« Soit le nom de l'élève est Michel et son prénom n'est pas Zoé, soit il est en PCSI 1 »

pourra directement être traduite par la requête :

```
SELECT * FROM eleve WHERE (nom = 'Michel' AND prenom != 'Zoé') OR classe = 'PCSI 1';
```

10.4.4 Opérations ensemblistes

Pour réaliser des opérations ensemblistes, il est d'abord nécessaire d'obtenir l'intégralité des tables grâce à une projection avec l'attribut *. Ensuite, on utilise les mots-clés `UNION`, `INTERSECT` ou `EXCEPT` pour réaliser les opérations d'union, d'intersection et de différence.

Par exemple, pour effectuer $R_1 \cap R_2$, on écrit :

```
SELECT * FROM R1 INTERSECT SELECT * FROM R2;
```

10.4.5 Renommage

Pour renommer un attribut, on décore une projection par le mot-clé `AS`.

Par exemple, on prend R une relation de schéma $(A_1, \cdots, A_n, C_1, \cdots, C_m)$ où l'on souhaite obtenir $\rho_{A_1,\cdots,A_n \leftarrow B_1,\cdots,B_n}(R)$.

On effectue :

```
SELECT A1 AS B1, ..., An AS Bn, C1, ..., Cm FROM R;
```

Il est donc nécessaire d'indiquer tous les attributs.

SAVOIR-FAIRE Traduire une question dans un langage de requête en respectant sa syntaxe

La construction de requêtes élaborées est facilitée par le fait que les commandes SQL traduisent simultanément plusieurs opérations. L'optimisation de ces requêtes est donc laissée à la charge du gestionnaire de bases de données.

Les requêtes SQL ne sont pas sensibles à la casse, mais il est d'usage de mettre les mots-clés en majuscules et les attributs en minuscules. Les valeurs des attributs non numériques sont écrites entre guillemets simples ou doubles. On note que :

- Toutes les requêtes de recherche commencent par le mot-clé `SELECT`, qui effectuera une projection en fin de requête.
- La relation sur laquelle on opère est précisée par `FROM`.
- Les requêtes SQL se terminent systématiquement par un point-virgule.

Attention au faux-ami : `SELECT` n'effectue pas une sélection, mais une projection.

Exercice 10.5 avec corrigé Traduire en langage SQL les requêtes de l'exercice 10.4.

1
```
SELECT filiere FROM classe;
```
On note que SQL ne fusionne pas les doublons dans une table. On peut forcer cette fusion à l'aide du mot-clé DISTINCT.

2
```
SELECT * FROM classe WHERE filiere='PCSI';
```

3
```
SELECT prenom FROM eleve WHERE classe=1 OR classe=3;
```

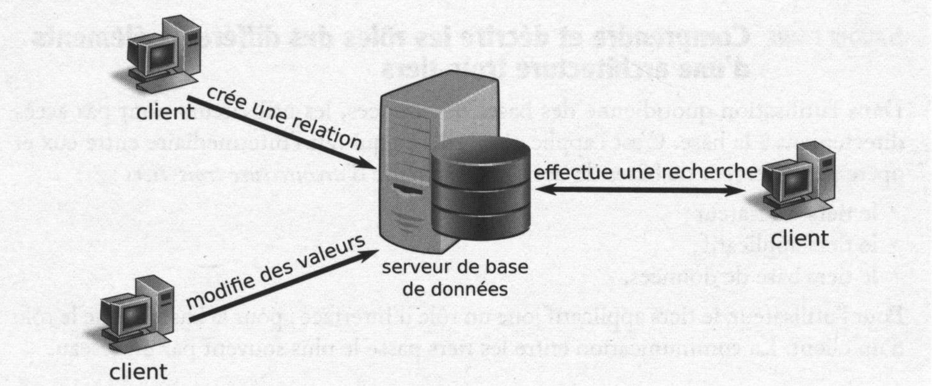
4
```
SELECT nom, note FROM eleve WHERE note<10;
```

10.5 Base de données et architecture logicielle

10.5.1 Architecture client-serveur

L'utilisation d'une base de données telle qu'on l'a vue, c'est-à-dire par un utilisateur effectuant des requêtes auprès d'un gestionnaire de bases de données, est appelée une *architecture client-serveur*.

L'utilisateur (qui est en réalité plus souvent un programme qu'un opérateur humain) est le client qui effectue des demandes auprès du serveur, lequel centralise les informations. La richesse de ce point de vue vient du fait que plusieurs clients peuvent accéder simultanément au serveur.



Ainsi, un client peut créer une relation, pendant qu'un deuxième modifie des valeurs d'une relation existante et que le troisième effectue une recherche. Une telle utilisation nécessite cependant de veiller à l'ordre dans lequel sont traitées les demandes. Par exemple si on effectue deux fois de suite la même recherche, elle peut produire des résultats différents si quelqu'un d'autre a modifié des valeurs s'y trouvant. La bonne approche consiste alors à grouper un ensemble de requêtes et à demander au serveur de les traiter de manière atomique, c'est-à-dire d'une traite. Un tel ensemble est appelé une *transaction*.

La communication entre le client et le serveur passe le plus souvent par une transmission d'informations sur un réseau, comme le réseau Internet. En effet, deux applications au sein d'un même ordinateur peuvent également communiquer selon ce mode client-serveur.

10.5.2 Architecture trois-tiers

L'architecture client-serveur est rarement mise en pratique telle quelle. En effet, il n'est pas nécessaire de traduire directement les opérations que l'on souhaite faire sur les données en termes de modèle relationnel.

On va prendre l'exemple d'un professeur devant rentrer ses notes de devoir surveillé. Il dispose le plus souvent d'une *application* fournie par son établissement, dans laquelle il peut créer une correction associée à sa classe et remplir les notes correspondantes. L'élève pourra, lui, consulter cette même application pour voir sa note. Enfin, l'administration pourra consulter les notes de l'ensemble des matières afin d'éditer des bulletins.

Ces différents types d'utilisateurs n'ont pas intérêt à accéder directement à la base de données. D'une part, on ne souhaite pas qu'un élève puisse voir l'ensemble des notes de ses camarades, ni que le professeur de mathématiques puisse créer un devoir de sciences industrielles. D'autre part, cela nécessiterait de connaître une grande partie du modèle relationnel, qui peut être très complexe.

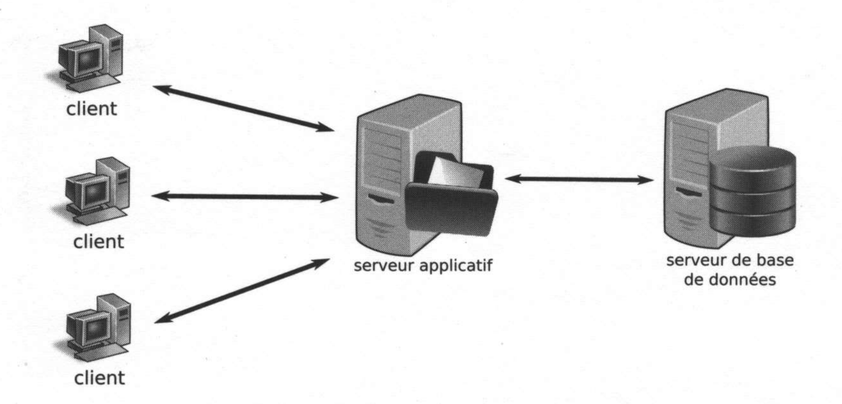
SAVOIR-FAIRE Comprendre et décrire les rôles des différents éléments d'une architecture trois-tiers

Dans l'utilisation quotidienne des bases de données, les utilisateurs n'ont pas accès directement à la base. C'est l'application utilisée qui fait l'intermédiaire entre eux et opère sur le serveur de bases de données. On parle d'*architecture trois-tiers* :

- le tiers utilisateur ;
- le tiers applicatif ;
- le tiers base de données.

Pour l'utilisateur, le tiers applicatif joue un rôle d'interface ; pour la base, il joue le rôle d'un client. La communication entre les tiers passe le plus souvent par un réseau.

Il est raisonnable de se demander si le tiers applicatif ne pourrait être fusionné avec un tiers base de données enrichi d'opérations *ad hoc*. En fait, dans un système complexe, un même serveur de bases de données est commun à une multitude d'architectures trois-tiers. On peut prendre l'exemple d'une entreprise. Une base de données pourra gérer l'annuaire du personnel ; le service Comptabilité aura alors une application dédiée pour éditer les payes des employés et le service Informatique une autre application pour gérer le parc informatique.



10.6 Exercices

Exercice 10.6 Proposer un domaine approprié pour des attributs représentant :

- une date ;
- un mot de passe ;
- un classement à un concours ;
- une note de musique.

Exercice 10.7 Proposer des schémas relationnels représentant respectivement :

- les animaux vivant dans un zoo ;
- les différents éléments chimiques avec des informations équivalentes à celles fournies par la table de Medeleïev.

Dans chaque cas, donner un exemple significatif de relation associée au schéma proposé.

Exercice 10.8 * Quelques propriétés mathématiques des opérations.

On considère une relation R de schéma S, $A, B \in S$, $a \in \mathbf{dom}(A)$, $b \in \mathbf{dom}(B)$ et $X \subset S$.

1 Montrer que $\sigma_{A=a}(\sigma_{B=b}(R)) = \sigma_{B=b}(\sigma_{A=a}(R))$.

2 Montrer que si l'on suppose $A \in X$, alors $\sigma_{A=a}(\pi_X(R)) = \pi_X(\sigma_{A=a}(R))$.
Est-ce toujours vérifié si $A \notin X$?

3 On suppose ici qu'on dispose d'une chaîne d'inclusion $X_1 \subset X_2 \subset \cdots \subset X_n \subset S$. Montrer que :

$$\pi_{X_1}(\pi_{X_2}(\cdots \pi_{X_n}(R)\cdots)) = \pi_{X_1}(R)$$

4 Soit $\bullet$ une des opérations ensemblistes $\{\cap, \cup, -\}$ et $R_1(S)$, $R_2(S)$ des relations de même schéma. Montrer que $\sigma_{A=a}(R_1 \bullet R_2) = \sigma_{A=a}(R_1) \bullet \sigma_{A=a}(R_2)$.

11

Base de données relationnelle

Dans ce chapitre, nous présentons des mécanismes permettant de répartir des données dans plusieurs relations liées entre elles. Nous étendons l'algèbre relationnelle avec des opérateurs capables de croiser les informations contenues dans plusieurs relations, ou de regrouper les informations des différentes valeurs d'une même relation. Nous traduisons également ces opérateurs supplémentaires dans un langage de requête.

11.1 Clé primaire

Lorsque l'on veut effectuer des rapprochements entre les valeurs de différentes relations, il est important de disposer d'un mécanisme efficace pour accéder à ces valeurs, tester leur existence ou encore les trier : c'est le rôle de la notion de *clé*.

11.1.1 Clé

Soit $R(S)$ une relation de schéma S et $K \subset S$. On dit que K est une *clé* pour R si et seulement si pour toutes valeurs $t_1, t_2 \in R$ telles que $t_1(K) = t_2(K)$ on a $t_1 = t_2$.

On considère par exemple la relation :

élève

Nom	Prénom
Meyer	Zoé
Michel	Nolwenn
Benoit	Paul
Michel	Zoé

On va examiner les différents sous-ensembles de $S = \{$**Nom, Prénom**$\}$. L'ensemble $\{$**Nom**$\}$ n'est pas une clé car les deux valeurs (Michel, Nolwenn) et (Michel, Zoé) ont même valeur pour cet attribut. De même, $\{$**Prénom**$\}$ n'est pas non plus une clé. En revanche, la paire $\{$**Nom, Prénom**$\}$ est une clé.

On remarque que le schéma d'une relation constitue toujours une clé pour celle-ci. Ainsi, une relation possède toujours au moins une clé. On recherche alors une bonne clé, c'est-à-dire une clé comportant le plus petit nombre possible d'attributs.

11.1.2 Clé primaire

Soit $R(S)$ une relation de schéma S et $A \in S$. On dit que A est une *clé primaire* pour $R(S)$ si et seulement si A est une clé pour R.

Une clé primaire est donc une très bonne clé car elle possède le nombre minimal d'attributs : un seul.

Autrement dit, A est une clé primaire si et seulement si pour toutes valeurs $e, e' \in R$ telles que $e \neq e'$ on a $e.A \neq e'.A$. Cela signifie que l'application :

$$\begin{array}{rcrcl} \pi_A & : & R & \mapsto & \mathsf{dom}(A) \\ & & e & \mapsto & e.A \end{array}$$

est injective.

Considérons la relation suivante :

livre

Titre	Auteur	Année-de-publication
Éléments	Euclide	-300
Traité du calcul différentiel	Euler	1755
Lettres à une princesse d'Allemagne	Euler	1768

L'attribut **titre** est une clé primaire, mais pas l'attribut **auteur**, car il existe deux livres écrits par Euler. On pourra l'indiquer sur un diagramme par un symbole de clé au niveau des attributs :



Il ne s'agit que d'une indication, car le schéma seul ne permet pas de garantir qu'un attribut soit une clé primaire. En effet, cela dépend des valeurs de la relation. Si l'on ajoute une valeur dont le titre est identique à l'un des trois titres déjà présents, **Titre** ne sera plus une clé primaire.

Ainsi, dans la relation *élève* présentée ci-avant il n'existe pas de clé primaire.

Donc, une relation possède toujours au moins une clé, mais ne possède pas forcément de clé primaire.

11.1.3 Lien entre deux tables

Le bénéfice d'une clé primaire A est que la donnée de $t.A$ suffit à identifier t de façon unique. Par exemple, sachant que, dans la relation *livre* ci-avant, **Titre** est une clé primaire, il est possible de faire référence à la première valeur par son titre : "Éléments".

Grâce à cela, on peut créer des liens entre plusieurs relations. On va imaginer par exemple la relation *emprunteur* de schéma relationnel :

((**nom**, Texte), (**livre-emprunté**, Texte))

La valeur ("Zoé", "Éléments") traduit donc le fait que Zoé a emprunté le livre Éléments. Une clé primaire sert ainsi à exprimer des liens entre relations.

Il est à noter cependant que le domaine de l'attribut **livre-emprunté** est trop peu restrictif : certaines valeurs de ce domaine (par exemple "Green Eggs and Ham") pourraient ne correspondre à aucune valeur de l'attribut **Titre** dans la relation *livre*.

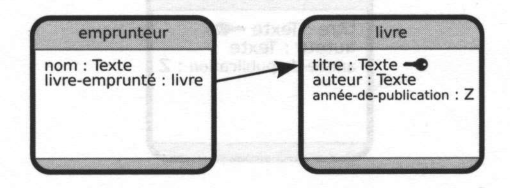
Pour aller plus loin Clé étrangère

On peut considérer que le domaine de **livre-emprunté** est constitué exactement des titres apparaissant dans la relation *livre*. On dit alors que *livre* est le domaine de l'attribut **livre-emprunté** et cet attribut est appelé une clé étrangère. On peut alors noter le schéma de manière plus précise :

((nom, Texte), (livre-emprunté, livre))

Les liens entre ces deux relations peuvent être représentés dans un diagramme, par une flèche depuis l'attribut vers la clé primaire.

Dans le cas présent, cela donne le diagramme suivant :



Exercice 11.1 Soit $R(S)$ une relation et $X \subset S$.
Montrer que $\#\pi_X(R) = \#R$ si et seulement si X est une clé.

Exercice 11.2 * La relation *élève* présentée p. 276 montre que si on représente des personnes dans une relation, en général l'attribut **Nom** ne constitue pas à lui seul une clé pour la relation ; il en va de même pour **Prénom**.

1. La paire d'attributs {**Nom**, **Prénom**} constitue-t-elle toujours une clé pour une relation de ce genre ?
2. Comment peut-on étendre le schéma relationnel pour assurer l'existence d'une clé ? Identifier des solutions utilisées dans des situations réelles et en évaluer les limites.
3. Identifier des situations réelles de relations représentant des personnes dans lesquelles un attribut du schéma relationnel est prévu pour être une clé *primaire*. Comment assure-t-on que cette clé sera bien primaire ?

11.2 Opérateurs complexes de l'algèbre relationnelle

Les opérateurs définis dans le chapitre précédent ne s'appliquent qu'à une seule relation ou, pour les opérateurs ensemblistes, à deux relations de même schéma. Dans une base de données, il est rare que deux relations aient le même schéma et, même dans ce cas, leur intersection ou leur union n'a pas forcément de sens. On définit donc d'autres opérateurs capables notamment de croiser les informations présentes dans plusieurs relations connexes.

11.2.1 Produit cartésien et division cartésienne

Comme l'union ou l'intersection, la notion de produit cartésien est issue des opérateurs ensemblistes.

Si $R(S)$ et $R'(S')$ sont deux relations de schémas disjoints, leur *produit cartésien* est :

$$R \times R' = \{ (v_1, \cdots, v_n, v'_1, \cdots, v'_m) \mid (v_1, \cdots, v_n) \in R \text{ et } (v'_1, \cdots, v'_m) \in R' \}.$$

Son schéma est :

$$S \uplus S' = (A_1, \cdots, A_n, B_1, \cdots, B_m) \text{ où } S = (A_1, \cdots, A_n) \text{ et } S' = (B_1, \cdots, B_m).$$

$R \times R'$ contient donc l'ensemble des possibilités d'association entre une valeur de R et une de R'. La notation $S \uplus S'$ rappelle qu'il ne s'agit pas seulement de prendre l'union des schémas, mais aussi de s'assurer qu'ils soient disjoints. Il est toujours possible de s'y ramener par l'intermédiaire d'un renommage des attributs présents dans les deux schémas.

Dans l'exemple qui suit, pour simplifier, on considère des relations ayant un seul attribut.

On prend une relation *élève* et une relation *enseignant* données par :

élève

Nom-élève
Meyer
Martin
Bernard

enseignant

Nom-enseignant
Durand
Petit

Leur produit cartésien est alors la relation :

élève × *enseignant*

Nom-élève	Nom-enseignant
Meyer	Durand
Martin	Durand
Bernard	Durand
Meyer	Petit
Martin	Petit
Bernard	Petit

La *division cartésienne* entre deux relations est alors définie de la même manière que la division euclidienne est définie par rapport à la multiplication entière.

La relation $R \div R'$ est la plus grande relation, vis-à-vis de l'inclusion, telle qu'il existe une relation R'' vérifiant :

$$[(R \div R') \times R'] \cup R'' = R$$

$$[(R \div R') \times R'] \cap R'' = \emptyset$$

On vérifiera que cette définition est bien fondée dans l'exercice 11.16.

On considère maintenant la relation R suivante :

R

Nom-élève	Nom-enseignant
Meyer	Durand
Bernard	Durand
Meyer	Petit
Martin	Petit
Bernard	Petit

Le quotient cartésien de R par *élève* est une sous-relation de *enseignant* :

$R \div$ *élève*

Nom-enseignant
Petit

Durand n'est pas associé à toutes les valeurs d'élèves dans R (l'élève Martin n'a pas Durand pour enseignant). Donc, il ne peut appartenir au quotient.

> **EN PRATIQUE La division cartésienne**
>
> L'opérateur de division cartésienne existe pour des raisons théoriques de complétude de l'algèbre relationnelle, c'est-à-dire pour s'assurer qu'il soit possible d'exprimer toutes les requêtes possibles. En pratique toutefois, il est absent des langages de requête.

11.2.2 Jointure

Un opérateur de jointure sert à recoller deux relations. On ne considère ici que l'opérateur de jointure symétrique, qui recolle deux relations de manière simple.

Soient $R(S)$ et $R'(S')$ deux relations de schémas disjoints, et $A \in S$, $A' \in S'$ tels que $\mathsf{dom}(A) = \mathsf{dom}(A')$. On note :

$$R\,[A = A']\,R' = \{\, e \in R \times R' \mid e.A = e.A' \,\} = \sigma_{A=A'}(R \times R')$$

appelée la *jointure symétrique* de S et S' selon (A, A').
Ainsi, $R\,[A = A']\,R'$ contient l'ensemble des valeurs obtenues en recollant une valeur de R et une valeur de R' dans le cas où les attributs A et A' y sont identiques.

On va considérer l'exemple des deux relations *livre* et *auteur* suivantes :

livre

Titre	Nom-auteur
Madame Bovary	Flaubert
Le père Goriot	Balzac (de)

auteur

Nom	Prénom
Flaubert	Gustave
Balzac (de)	Honoré
Proust	Marcel

On va réaliser la jointure symétrique selon (**Nom-auteur, Nom**), ce qui donnera une nouvelle relation dont le schéma sera (**Titre, Nom-auteur, Nom, Prénom**).

Chaque livre sera alors associé à l'intégralité des informations stockées dans la relation *auteur*. Une telle construction est souvent utile comme étape intermédiaire pour construire d'autres relations, comme on va le voir dans la suite.

Ici, on obtient :

livre [*Nom-auteur* = *Nom*] *auteur*

Titre	Nom-Auteur	Nom	Prénom
Madame Bovary	Flaubert	Flaubert	Gustave
Le père Goriot	Balzac (de)	Balzac (de)	Honoré

Il est à noter que la présence des deux attributs A et A' est redondante après jointure. On peut éliminer cette redondance avec une projection :

$\pi_{Titre,Nom,Prénom}(livre\ [Nom\text{-}auteur = Nom]\ auteur)$

Titre	Nom	Prénom
Madame Bovary	Flaubert	Gustave
Le père Goriot	Balzac (de)	Honoré

Dans cet exemple, l'attribut Nom est une clé primaire pour *auteur* ; donc, en réalisant la jointure, on sait que l'on va récupérer un unique auteur. Ce cas est le plus courant, car il permet effectivement d'*éclater* un attribut faisant référence à une clé primaire pour lui substituer l'intégralité de la valeur à laquelle elle fait référence. Ici, on remplace le simple nom de l'auteur par la valeur correspondante dans *auteur*.

Cependant, rien n'oblige à considérer des clés primaires pour réaliser des jointures. On va modifier *auteur* afin que Nom ne soit plus une clé primaire :

auteur'

Nom	Prénom
Flaubert	Gustave
Flaubert	Jean-Claude
Balzac (de)	Honoré
Proust	Marcel

Le résultat de la jointure précédente donnera alors :

$livre\ [Nom\text{-}auteur = Nom]\ auteur'$

Titre	Nom-Auteur	Nom	Prénom
Madame Bovary	Flaubert	Flaubert	Gustave
Madame Bovary	Flaubert	Flaubert	Jean-Claude
Le père Goriot	Balzac (de)	Balzac (de)	Honoré

Ainsi, le titre *Madame Bovary* se trouve associé aux deux auteurs nommés Flaubert dans *auteur'*.

On étend alors logiquement la notion de jointure au cas de recollement de plusieurs couples d'attributs. On pourra noter par exemple :

$$livre\ [\textbf{Nom-auteur} = \textbf{Nom}, \textbf{Prenom-auteur} = \textbf{Prenom}]\ auteur'$$

SAVOIR-FAIRE Concevoir une base constituée de plusieurs relations et utiliser les jointures symétriques pour effectuer des requêtes croisées

On conçoit une base de données constituée de plusieurs relations lorsqu'on veut isoler des informations concernant différentes entités (les livres et les auteurs dans l'exemple) ; chaque relation est propre à une de ces entités et ne contient que les informations qui la caractérisent.

Si les valeurs d'une relation R_1 doivent faire référence à celles d'une relation R_2, on prévoit dans R_2 un attribut A_2, si possible une clé, auquel la relation R_1 pourra faire référence par le biais d'un de ses propres attributs A_1, qui sera une clé étrangère.

La jointure symétrique fait correspondre à chaque valeur de R_1 toutes les informations de la valeur qui lui correspond dans R_2, ce qui autorise à effectuer une sélection selon ces informations.

Une même base de données contient souvent plus de deux relations, éventuellement reliées deux à deux par le biais de divers attributs. On peut alors écrire des requêtes élaborées au moyen de plusieurs jointures symétriques.

Exercice 11.3 avec corrigé On va considérer ici un ensemble de relations utiles pour gérer un complexe hôtelier. Celui-ci est composé de différents bâtiments, identifiés par leur nom et leur nombre d'étoiles. Ils sont représentés dans la relation *batiment* :

batiment

nom	etoiles
Rose	3
Jasmin	2
Lys	3

Les chambres comportent chacune un numéro, un nom de bâtiment et un nombre de fenêtres. Elles sont représentées dans la relation *chambre* :

chambre

numero	batiment	fenetres
1	Rose	2
2	Rose	1
3	Rose	1
1	Jasmin	1
2	Jasmin	0
3	Jasmin	1
4	Jasmin	1
1	Lys	3
2	Lys	2
3	Lys	2

Certaines chambres possèdent deux lits. On sépare donc les lits dans une autre relation, où ils comportent chacun un identifiant numérique unique au sein du complexe, un numéro de chambre, ainsi que le bâtiment correspondant. Ils sont représentés dans la relation *lit* :

lit

idlit	chambre	batlit
1	1	Rose
2	1	Rose
3	2	Rose
4	3	Rose
5	1	Jasmin
6	2	Jasmin
7	2	Jasmin
8	3	Jasmin
9	3	Jasmin
10	4	Jasmin
11	1	Lys
12	2	Lys
13	3	Lys

Enfin, les nuitées sont identifiées par le nom du client, l'identifiant du lit et par la date. Elles sont représentées dans la relation *nuitee* :

nuitee

client	lit	date
Lennon	1	15-08-1969
McCartney	8	18-08-1969
Starr	3	03-07-1969
Harrison	2	01-08-1969
Page	10	05-08-1969
Plant	1	13-08-1969
Jones	11	05-08-1969
Bonham	7	02-08-1969
Townshend	1	08-08-1969

Pour chacune des recherches suivantes, on indiquera une décomposition dans l'algèbre relationnelle, ainsi que le résultat obtenu.

1 Obtenir le nom des clients ayant séjourné dans le bâtiment Jasmin.
2 Obtenir le nom des clients ayant séjourné dans un bâtiment 3 étoiles.
3 Obtenir le nom des clients ayant séjourné dans une chambre ayant au moins 2 fenêtres.

Dans tous les cas, il faudra finir par une projection $\pi_{\textbf{client}}$. *On omet donc celle-ci, afin de se concentrer sur les jointures et autres opérations.*

1 *L'information du bâtiment est accessible depuis la relation* lit*; on effectue une jointure entre les relations* nuitee *et* lit*, puis on sélectionne le bâtiment voulu :*

$$\sigma_{\textbf{batlit}=\text{Jasmin}}(\text{nuitee }[\textbf{lit} = \textbf{idlit}]\text{ lit})$$

On obtient les noms : McCartney, Page et Bonham.

2 *Ici, il est nécessaire d'effectuer une jointure supplémentaire avec la relation* batiment*. Dans la mesure où les jointures sont associatives, on peut les effectuer dans n'importe quel ordre. On omet donc les parenthèses pour plus de clarté :*

$$\sigma_{\textbf{etoiles}=3}(\text{nuitee }[\textbf{lit} = \textbf{idlit}]\text{ lit }[\textbf{batlit} = \textbf{nom}]\text{ batiment})$$

On obtient les noms : Lennon, Starr, Harrison, Plant, Jones et Townshend.

3 *Pour cette requête, on remarque tout d'abord qu'il n'existe pas de clé primaire pour la relation* chambre*. Il faudra donc effectuer une jointure sur la clé* $\{\textbf{numero}, \textbf{batiment}\}$ *à l'aide de deux conditions de recollement :*

$$\sigma_{\textbf{fenetres}\geq 2}(\text{nuitee }[\textbf{lit} = \textbf{idlit}]\text{ lit }[\textbf{batlit} = \textbf{batiment}, \textbf{chambre} = \textbf{numero}]\text{ chambre})$$

On obtient les noms : Lennon, Harrison, Plant, Jones et Townshend.

EN PRATIQUE La jointure dans les gestionnaires de bases de données

D'un point de vue théorique, on pourrait définir une algèbre relationnelle sans cet opérateur de jointure, puisqu'il s'exprime comme une composition d'un produit cartésien et d'une sélection.

Concrètement, ce serait une très mauvaise idée de programmer les jointures par le biais de ces deux autres opérations : si les relations R et R' contiennent respectivement n et n' valeurs, le produit cartésien $R \times R'$ construit une relation de $n \times n'$ valeurs, qu'il faut ensuite parcourir pour effectuer la sélection, d'où un coût quadratique.

Avec les tailles courantes des bases de données, un tel coût est impraticable et, de plus, la relation $R \times R'$ a peu de chances de tenir dans la mémoire vive disponible. Les gestionnaires de bases de données disposent d'algorithmes efficaces pour effectuer la jointure de deux tables de taille n avec une complexité en $O(n \log n)$.

11.2.3 Agrégation

Le dernier concept qu'on va présenter est assez complexe, mais très expressif. On va imaginer que l'on dispose de la relation suivante :

relevé

Classe	Élève	Note
MPSI	Meyer	17,5
PCSI	Martin	7,75
MPSI	Bernard	9,25
PCSI	Robert	14,0
PCSI	Dubois	11,5

L'agrégation va servir à regrouper les élèves d'une même classe (ce groupe de valeurs est appelé un *agrégat*) et à effectuer une opération sur chacun des agrégats. Ici par exemple, on pourrait calculer la moyenne sur chaque classe, ce qui produit la relation suivante :

$_{Classe}\gamma_{moyenne(Note)}(relevé)$

Classe	*moyenne*(Note)
MPSI	13.38
PCSI	11.08

Avant de définir formellement cette opération, il faut définir une notion de fonction pouvant être appliquée pour un nombre quelconque d'arguments, indépendamment de leur ordre.

POUR ALLER PLUS LOIN Fonctions d'agrégation

Calculer la moyenne d'un ensemble d'élèves ne dépend pas de l'ordre dans lequel on les énumère. On dit que la fonction *moyenne* est *symétrique*.

Plus généralement, une application $f : D^n \to D$, où D est un ensemble, est dite *symétrique* si et seulement si $\forall x_1, \cdots, x_n \in D$ et pour tout $i, j \in [\![1; n]\!]$ avec $i < j$ on a :

$$f(x_1, \cdots, x_n) = f(x_1, \cdots, x_{i-1}, x_j, x_{i+1}, \cdots, x_{j-1}, x_i, x_j, \cdots, x_n)$$

Autrement dit, $f(x_1, \cdots, x_n)$ ne change pas de valeur en échangeant x_i et x_j.

Pour pouvoir appliquer une fonction, comme la somme, à des relations de tout cardinal, il faut considérer des suites d'applications. Une suite $(f_n)_{n\in\mathbb{N}}$ d'applications symétriques de D^n dans X est appelée une *fonction d'agrégation*. Par abus de notation, on notera directement

$$f(x_1, \cdots, x_n) = f_n(x_1, \cdots, x_n)$$

On ne considère ici que cinq fonctions d'agrégation :

- *comptage*, qui associe à tout n-uplet son nombre d'éléments ;
- *max* (resp. *min*), qui associe à tout n-uplet le plus grand (resp. le plus petit) de ses éléments ;
- *moyenne* (resp. *somme*), qui calcule la moyenne (resp. la somme) de chaque n-uplet.

Soit $R(S)$ une relation, $A \in S$ et f une fonction d'agrégation. On note $f(R.A)$ le résultat de l'application de la fonction f au n-uplet des valeurs de R pour l'attribut A.

Soient $R(S)$ une relation, $A_1, \cdots, A_n, B_1, \cdots, B_m \in S$ et $f_1, \cdots, f_m$ des fonctions d'agrégation. On note ${}_{A_1,\ldots,A_n}\gamma_{f_1(B_1),\ldots,f_m(B_m)}(R)$ la relation obtenue :

- en regroupant les valeurs de R qui sont identiques sur les attributs $A_1, \cdots, A_n$;
- et en définissant de nouveaux attributs $f_i(B_i)$ pour ces valeurs regroupées, pour tout $i \in [\![1; m]\!]$, par application de la fonction d'agrégation f_i sur chacun de ces agrégats sur l'attribut B_i.

Dans le cas particulier où l'on n'effectue pas de regroupement, on note $\gamma_{f_1(B_1),\ldots,f_m(B_m)}(R)$ l'opération.

Lorsque l'on n'effectue que le regroupement, on la note ${}_{A_1,\ldots,A_n}\gamma(R)$.

Attention Sélection en amont et en aval d'une agrégation

On considère une opération de la forme :

$$\sigma_{P_2} \circ {}_{A_1,\cdots,A_n}\gamma_{f_1(B_1),\ldots,f_m(B_m)} \circ \sigma_{P_1}$$

Le rôle des deux sélections est très différents :

- La sélection σ_{P_1} est effectuée avant le regroupement en agrégat et limite ainsi les valeurs considérées pour l'agrégation.
- La sélection σ_{P_2} porte, elle, sur les agrégats décorés des fonctions évaluées.

Exercice 11.4 avec corrigé On considère la relation suivante :

relevé

Filière	Numéro	Élève	Note
MPSI	1	Meyer	17,5
PCSI	2	Martin	7,75
MPSI	1	Bernard	9,25
PCSI	1	Robert	14,0
PCSI	2	Dubois	11,5
MPSI	1	Lemaire	7,25
PCSI	1	Albert	13,0
PCSI	1	Garcia	16,5
PCSI	2	Richard	12,5
MPSI	2	Petit	15,5
PCSI	1	Simon	10,5

Traduire les requêtes suivantes en opérations de l'algèbre relationnelle.

1 Calculer la moyenne des PCSI et celle des MPSI.
2 Calculer la moyenne de chaque classe.
3 Calculer la moyenne de la PCSI 2 et celle de la MPSI 2.
4 Sélectionner les classes dont la moyenne est supérieure ou égale à douze.

1

$_{\textbf{Filière}}\gamma_{moyenne(\textbf{Note})}$

Filière	$moyenne$(***Note***)
MPSI	*12.375*
PCSI	*12.25*

2

$_{\textbf{Filière,Numéro}}\gamma_{moyenne(\textbf{Note})}$

Filière	***Numéro***	$moyenne$(***Note***)
MPSI	*1*	*11.33*
MPSI	*2*	*15.5*
PCSI	*1*	*13.5*
PCSI	*2*	*10.6*

3 *Ici, on peut le faire avec une sélection en amont :*

$_{\textbf{Filière,Numéro}}\gamma_{moyenne(\textbf{Note})} \circ \sigma_{\textbf{Numéro}=2}$

Filière	***Numéro***	$moyenne$(***Note***)
MPSI	*2*	*15.5*
PCSI	*2*	*10.6*

ou en aval :

$\sigma_{\textbf{Numéro}=2} \circ {}_{\textbf{Filière,Numéro}}\gamma_{moyenne(\textbf{Note})}$

Filière	***Numéro***	$moyenne$(***Note***)
MPSI	*2*	*15.5*
PCSI	*2*	*10.6*

Il faut toujours privilégier les sélections en amont, car elles limitent le nombre de valeurs à regrouper pour l'agrégation.

4 *Ici, on ne peut effectuer la sélection qu'en aval :*

$\sigma_{moyenne(\textbf{Note})\geq 12} \circ {}_{\textbf{Filière,Numéro}}\gamma_{moyenne(\textbf{Note})}$

Filière	***Numéro***	$moyenne$(***Note***)
MPSI	*2*	*15.5*
PCSI	*1*	*13.5*

11.2.4 Composition de requêtes complexes

Avec ces opérateurs supplémentaires, on peut exprimer des requêtes plus élaborées, faisant notamment intervenir plusieurs relations. Leur écriture est alors plus délicate. En particulier, la question de l'ordre dans lequel composer les différentes opérations devient alors une vraie question de recherche.

SAVOIR-FAIRE Traduire dans le langage de l'algèbre relationnelle des requêtes complexes

Pour traduire une requête complexe, une possibilité (pas toujours optimale cependant) est de commencer par les opérations les plus simples, que l'on combine ensuite pour former des opérations plus complexes. Si nécessaire, on peut aller jusqu'à décomposer la requête en sous-requêtes qu'on écrit séparément.

Lorsqu'une requête concerne des informations réparties dans plusieurs relations, on effectue une jointure pour les rassembler en une seule relation.

Lorsqu'une requête demande d'effectuer un calcul sur un ensemble de valeurs, on identifie la fonction correspondante et on effectue une agrégation.

Exercice 11.5 avec corrigé On reprend les relations de l'exercice 10.4 :

classe

id	Filière	Numéro	Professeur
1	MPSI	1	Euclide
2	MPSI	2	Turing
3	PCSI	1	Horner
4	PCSI	2	Euler

élève

Nom	Prénom	Classe	Note
Meyer	Romain	1	9,25
Martin	Paul	2	7,75
Robert	Marie	4	12,0
Michel	Lucile	2	11,5
Bernard	Sylvie	1	17,5
Martin	Romain	3	14,0
Meyer	Pierre	1	10,0
Dubois	Camille	3	11,5

Ici, on est dans le cas classique où **id** est une clé primaire pour *classe*, et *élève* possède une clé étrangère vers *classe*. Traduire les requêtes suivantes en opérations de l'algèbre relationnelle.

1 Calculer la moyenne des élèves de la classe 3.
2 Obtenir les élèves scolarisés en filière MPSI.
3 Déterminer la classe ayant la meilleure moyenne.

1 *On sélectionne les élèves, puis on effectue une agrégation avec la fonction $moyenne$:*

$$\gamma_{moyenne(\textbf{Note})}(\sigma_{\textbf{Classe}=3}(\text{élève}))$$

2 *Il faut effectuer une jointure pour connaître la filière de chaque élève :*

$$\pi_{\textbf{Nom},\textbf{Prénom},\textbf{Classe},\textbf{Note}}(\sigma_{\textbf{Filière}=MPSI}(\text{élève } [\textbf{Classe} = \textbf{id}] \text{ classe}))$$

3 *On décompose la requête en sous-requêtes.*
Pour calculer la moyenne des différentes classes, on peut utiliser une agrégation avec la fonction $moyenne$ en regroupant les élèves selon leur classe :

$$R = {}_{\textbf{Classe}}\gamma_{moyenne(\textbf{Note})}(\text{élève})$$

On effectue une nouvelle agrégation (cette fois sans regroupement) pour déterminer la moyenne maximale :

$$R' = \gamma_{max(moyenne(\textbf{Note}))}(R)$$

Puis on détermine l'identifiant de la classe ayant cette moyenne :

$$R'' = \sigma_{moyenne(\textbf{Note})=R'}(R)$$

Il reste alors à sélectionner l'entrée correspondante dans classe*, ce qu'on fait au moyen d'une jointure :*

$$R'' \ [\textbf{Classe} = \textbf{id}] \text{ classe}$$

On peut terminer par une projection si on ne souhaite pas conserver toutes les informations concernant cette classe. L'opération globale est donc ici composée de beaucoup d'opérations simples.

11.3 Traduction en langage SQL

Les opérateurs complexes se traduisent également par des requêtes de recherche commençant par la commande SELECT.

11.3.1 Jointure

La jointure simple $R\ [A = B]\ R'$ s'écrit :

```
SELECT * FROM R JOIN R' ON A=B;
```

Là encore, il s'agit d'une extension de la commande SELECT et on peut donc directement la combiner avec une projection, voire une sélection.

Le produit cartésien s'exprime simplement comme une jointure sans condition ON.

Dans le cas où l'on souhaite réaliser plusieurs jointures, on pourra utiliser un produit cartésien suivi d'une condition WHERE. Ainsi, la jointure $R\ [A = B]\ R'\ [C = D]\ R''$ s'écrit :

```
SELECT * FROM R,R',R'' WHERE A=B AND C=D;
```

Exercice 11.6 avec corrigé Traduire dans le langage SQL les recherches effectuées dans l'exercice 11.3.

1
```
SELECT client FROM nuitee JOIN lit ON lit=idlit WHERE batlit = 'Jasmin';
```
ou encore
```
SELECT client FROM nuitee, lit WHERE lit=idlit AND batlit = 'Jasmin';
```

2
```
SELECT client FROM nuitee, lit, batiment
              WHERE lit=idlit AND batlit=batiment AND etoiles=2;
```

3
```
SELECT client FROM nuitee, lit, chambre
              WHERE lit=idlit AND batlit=batiment AND chambre=numero AND fenetres>=2;
```

11.3.2 Application simple d'une fonction d'agrégation

Pour calculer $\gamma_{f(A)}(R)$, on utilise là encore une projection spéciale :

```
SELECT f(A) FROM R;
```

Le tableau suivant fournit la correspondance entre les fonctions d'agrégation que l'on a vues et celles de SQL :

Algèbre relationnelle	SQL
comptage	COUNT
max	MAX
min	MIN
somme	SUM
moyenne	AVG

Il est possible de coupler ce calcul à une projection, comme on l'a vu précédemment, par exemple :

```
SELECT max(note) AS note FROM eleve WHERE classe=1;
```

Le résultat renvoyé par l'agrégation étant de type numérique, il est possible de l'utiliser dans des comparaisons. Donc, pour obtenir les élèves ayant obtenu plus que la moyenne de la classe, on écrit :

```
SELECT * FROM eleves WHERE note >= (SELECT avg(note) FROM eleves);
```

Exercice 11.7 avec corrigé Traduire en langage SQL les requêtes écrites à l'exercice 10.4.

1
```
SELECT avg(note) FROM eleve WHERE classe=3;
```

2
```
SELECT nom, prenom, classe, note FROM eleve JOIN classe ON classe=id WHERE filiere='MPSI';
```

3 *La première requête est la suivante, dans laquelle on renomme $moyenne$(**Note**) pour pouvoir y faire référence par la suite :*
```
SELECT classe, avg(note) AS moyenne FROM eleve GROUP BY classe;
```

On peut ensuite l'imbriquer dans une autre requête, mais ceci se révèle vite malcommode, surtout dans ce cas où le résultat de cette requête doit être utilisé deux fois (une fois pour déterminer le maximum, une autre fois pour retrouver la classe correspondante).
En pratique, on préfère stocker les résultats des requêtes intermédiaires dans des tables temporaires au moyen de la commande `CREATE VIEW` *pour leur appliquer les requêtes suivantes, mais cela dépasse le cadre de cet ouvrage.*

11.3.3 Agrégation

Pour calculer $_{A_1,\ldots,A_n}\gamma_{f_1(B_1),\ldots,f_m(B_m)}$ on écrit :

```
SELECT A1, ..., An, f1(B1), ..., fm(Bm) FROM R GROUP BY A1, ..., An;
```

On remarque qu'il est également possible de coupler ce calcul à une projection pour ajouter d'autres attributs.

Pour effectuer une sélection en amont il faut utiliser une condition `WHERE` *avant* l'instruction `GROUP BY`.

Pour réaliser une sélection en aval, il faut utiliser une syntaxe spécifique de condition en utilisant l'instruction `HAVING` *après* l'instruction `GROUP BY`.

Ainsi l'expression :

$$(\sigma_{P_2} \circ {}_{A_1,\cdots,A_n}\gamma_{f_1(B_1),\ldots,f_m(B_m)} \circ \sigma_{P_1})(R)$$

se traduit par la requête :

```
SELECT A1, ..., An, f1(B1), ..., fm(Bm) FROM R WHERE P1 GROUP BY A1, ..., An HAVING P2;
```

SAVOIR-FAIRE Choisir entre une sélection en amont (`WHERE`) et une sélection en aval (`HAVING`)

Dans la mesure du possible, il faut toujours réaliser les sélections en amont, car cela limite le nombre de valeurs à considérer dans l'agrégation.
Cependant, lorsque l'on désire sélectionner en fonction du résultat des fonctions d'agrégation, il est obligatoire de le faire en aval à l'aide d'une instruction `HAVING`.

Exercice 11.8 avec corrigé Traduire en langage SQL les requêtes écrites à l'exercice 11.4.

1
```
SELECT filiere, avg(note) FROM releve GROUP BY filiere;
```

2
```
SELECT filiere, numero, avg(note) FROM releve GROUP BY filiere, numero;
```

3
```
SELECT filiere, numero, avg(note) FROM releve WHERE numero = 2, GROUP BY filiere, numero;
```

4
```
SELECT filiere, numero, avg(note) FROM releve
    GROUP BY filiere, numero HAVING avg(note) >= 12;
```

11.4 Exercices

Exercice 11.9 Pour chacun des schémas relationnels proposés dans l'exercice 10.7, peut-on espérer qu'il existe une clé primaire ? Si oui, laquelle ?

Est-il possible d'adapter le schéma relationnel pour qu'il comporte une clé primaire qui ait du sens ?

Exercice 11.10 Pourquoi n'est-il en général pas judicieux de construire un schéma relationnel dans lequel plusieurs attributs sont des clés primaires ?

Dans un tel cas, comment peut-on réorganiser la base de données à l'aide de la notion de clé étrangère ?

Exercice 11.11 * Un loueur de voitures et d'utilitaires souhaite informatiser la gestion de son entreprise. La base de données devra notamment stocker des informations sur :

- les types de véhicules proposés : volume utile, coût d'entretien annuel, tarif de location...
- les véhicules effectivement possédés par l'entreprise : type, marque, immatriculation, est-il en cours de location ou non...
- les clients : coordonnées, véhicule loué, dates de location...

Proposer un ensemble de schémas relationnels, de clés primaires et étrangères permettant de réaliser la gestion de cette entreprise.

Exercice 11.12 Opérations ensemblistes et clés. Soit $R(S)$ et $R'(S)$ deux relations de même schéma, et $K \subset S$ une clé pour R et pour R'.

Montrer que S est une clé pour $R \cap R'$. Déterminer un contre-exemple afin d'affirmer que K n'est pas nécessairement une clé pour $R \cup R'$.

Exercice 11.13 Application d'opérations sur des relations. On considère ici les relations suivantes :

R_1

id	**Nombre**	**Premier**
1	2	1
2	3	1
3	4	0
4	5	1
5	8	0
6	11	1
7	15	0
8	19	1

R_2

Diviseur	**Multiple**
2	4
3	6
4	8
5	10
6	12

1 Déterminer les résultats des opérations suivantes :
   - $\sigma_{\textbf{Premier}=1}(R_1)$
   - R_1 [**Nombre = Diviseur**] R_2
   - $somme(R_1.\textbf{Nombre})$
   - $\pi_{\textbf{Premier}}(R_1)$

2 Écrire les opérations, éventuellement composées, permettant d'obtenir les résultats suivants :
   - la somme des nombres premiers (ceux dont l'attribut **Premier** vaut 1) ;
   - la somme des nombres présents dans R_2 dont l'attribut **Diviseur** est premier ;
   - la vérification qu'un nombre indiqué comme premier n'est pas une valeur de l'attribut **Multiple** de R_2.

3 Traduire dans le langage SQL chacune des opérations vues dans les questions précédentes.

Exercice 11.14 Arbre généalogique. On considère ici la relation Valois(S), où :

$$S = (\,(\textbf{id}, \mathbb{N}), (\textbf{Nom}, \textbf{Texte}), (\textbf{Parent}, \mathbb{N})\}\,)$$

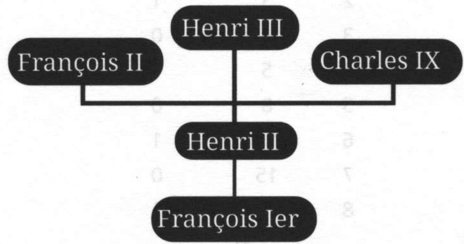
Valois

id	**Nom**	**Parent**
1	François Ier	0
2	Henri II	1
3	François II	2
4	Charles IX	2
5	Henri III	2

avec **id** une clé primaire pour *Valois* et **Parent** un attribut définissant un lien de *Valois* vers elle-même.

On convient qu'une valeur **Parent** de 0 exprime le fait qu'elle n'a pas de parent dans la relation.

Cette relation représente la généalogie de quelques rois de France, ce que l'on représente usuellement par l'arbre généalogique suivant :



1 Écrire les opérations, éventuellement composées, fournissant les informations suivantes :
   - les fils d'Henri II ;
   - le père de Charles IX ;
   - les descendants de François Ier ;
   - les ancêtres de Henri III.

2 Comment peut-on déterminer la personne à la racine de l'arbre ?

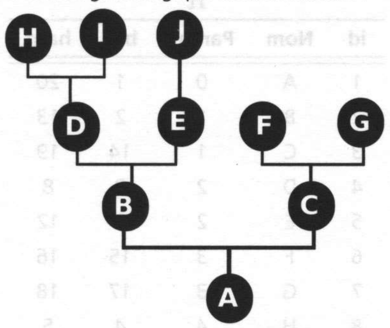
3 Comment peut-on déterminer les personnes n'ayant aucun enfant ?

4 Estimer le nombre de sélections que l'on doit faire pour récupérer les ancêtres dans un arbre généalogique quelconque.

5 Estimer le nombre de sélections que l'on doit faire pour récupérer les descendants dans un arbre généalogique quelconque.

L'exercice suivant présente une manière plus efficace de représenter de tels arbres.

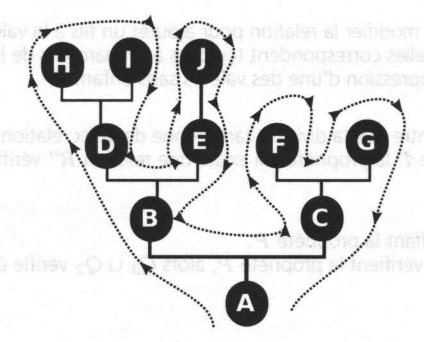
Exercice 11.15 ** Arbres à parcours précalculé. On considère ici une relation similaire à celle de l'exercice précédent. Elle correspond à l'arbre généalogique abstrait suivant :



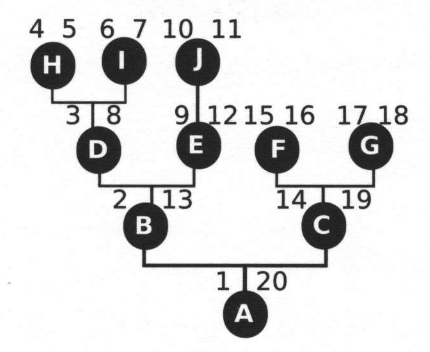
Cependant, on va ajouter à chaque valeur deux entiers *bas* et *haut* correspondant au premier et au dernier moment où l'on rencontre une personne en effectuant le parcours suivant :

1 On commence par la racine comme personne courante.
2 Si la personne courante a des enfants, on désigne successivement chacun d'eux comme personne courante et on continue le parcours à partir de celui-ci.
3 Sinon on s'arrête.

Un tel parcours est représenté sur le diagramme suivant :



Sur celui-ci sont indiquées les deux valeurs *bas* (à gauche) et *haut* (à droite) :



Cet arbre enrichi de ces deux nouvelles informations est représenté dans la relation suivante :

R

id	Nom	Parent	bas	haut
1	A	0	1	20
2	B	1	2	13
3	C	1	14	19
4	D	2	3	8
5	E	2	9	12
6	F	3	15	16
7	G	3	17	18
8	H	4	4	5
9	I	4	6	7
10	J	5	10	11

1 Montrer que dans une telle représentation, il est possible d'obtenir les descendants d'une valeur avec une seule sélection.
2 Même question pour les ancêtres.
3 Parmi les ancêtres obtenus, comment peut-on distinguer le père ? En déduire que l'attribut **Parent** est redondant.
4 Indiquer comment on peut modifier la relation pour ajouter un fils à la valeur E tout en préservant les valeurs **bas** et **haut** afin qu'elles correspondent toujours à un parcours de l'arbre.
5 Même question pour la suppression d'une des valeurs sans enfant.

Exercice 11.16 ** On démontre que la division cartésienne de deux relations R et R' est bien définie. Si Q est une relation, on appelle P la propriété « il existe une relation R'' vérifiant $(Q \times R') \cup R'' = R$ et $(Q \times R') \cap R'' = \emptyset$ ».

Démontrer que :

1 Il existe une relation Q vérifiant la propriété P.
2 Si deux relations Q_1 et Q_2 vérifient la propriété P, alors $Q_1 \cup Q_2$ vérifie également P.

Conclure.

Cinquième partie

Algorithmique et programmation avancées

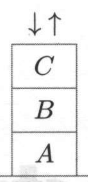
Cette partie couvre, avec la section sur les fonctions récursives du chapitre 5, le programme de deuxième année. Nous y montrons qu'il existe d'autres structures de données telles que la pile (chapitre 12) et nous y comparons plusieurs algorithmes de tri (chapitre 13) du point de vue de leurs complexités.

12

Structure de pile

Une des problématiques importantes de l'informatique est le stockage des données. Pour traiter efficacement ces dernières, il faut les ranger de manière adéquate, de même que l'on met de l'ordre dans ses affaires pour s'y retrouver plus facilement. L'objet informatique qui stocke des valeurs en mémoire s'appelle une structure de données, *qui est caractérisée par les opérations qu'elle permet et le coût de ces opérations. On peut ainsi vouloir connaître le nombre d'éléments qu'elle contient, accéder à un élément donné, en parcourir tous les éléments, etc. Le tableau est la seule structure de données des langages de programmation présentée jusqu'alors. Nous allons maintenant aborder les piles.*

Dans ce chapitre, on présente la structure de *pile*. Elle correspond exactement à l'image traditionnelle d'une pile de cartes ou d'assiettes posée sur une table. En particulier, on ne peut accéder qu'au dernier élément ajouté, qu'on appelle le *sommet* de la pile. Ainsi, si on a ajouté successivement A, puis B, puis C dans une pile, on se retrouve dans la situation suivante :



C est empilé sur B, lui-même empilé sur A. On peut soit retirer C de la pile (on dit qu'on « dépile » C), soit ajouter un quatrième élément D (on dit qu'on « empile » D). Si on veut accéder à l'élément A, il faut commencer par dépiler C, puis B. L'image associée à une pile est donc « dernier arrivé, premier sorti » (en anglais *last in, first out*, parfois abrégé en LIFO).

POUR ALLER PLUS LOIN Structure de file

On peut bien évidemment aussi imaginer une structure où, comme dans une file d'attente, les éléments sortent dans leur ordre d'arrivée, le premier élément sorti correspondant à l'élément le plus anciennement arrivé (en anglais *first in, first out*, parfois abrégé en FIFO). On parle alors de structure de *file*. De mise en œuvre légèrement plus délicate que celle de la structure de pile présentée dans ce chapitre, la structure de file ne figure pas au programme.

12.1 Opérations caractérisant une structure de pile

Avant de se pencher sur les différentes façons de réaliser une structure de pile, il faut définir l'ensemble des opérations qu'une telle réalisation doit pouvoir fournir, quelle qu'elle soit. Les trois opérations principales sont réalisées par les fonctions `creer_pile`, `depiler` et `empiler`.

- `creer_pile(c)` renvoie une nouvelle pile de capacité `c`, initialement vide.
- `depiler(p)` dépile et renvoie le sommet de la pile `p`.
- `empiler(p, v)` empile la valeur `v` sur la pile `p`.

Les opérations `depiler` et `empiler` modifient le contenu de la pile passée en argument. Voici une illustration de l'utilisation de ces trois opérations :

```
p = creer_pile(10)
empiler(p, A)
```

A

```
empiler(p, B)
empiler(p, C)
```

C
B
A

```
depiler(p)
```

B
A

D'autres opérations sont disponibles, mais ne modifient pas la pile :

- `taille(p)` renvoie le nombre d'éléments contenus dans la pile *p*.
- `est_vide(p)` indique si la pile *p* est vide.
- `sommet(p)` renvoie le sommet de la pile *p*, sans modifier *p*.

SAVOIR-FAIRE Choisir un type de données en fonction d'un problème à résoudre

La pile est une structure de données appropriée quand :

- On veut stocker des éléments dont le nombre est variable, a fortiori dont le cardinal maximum est inconnu à l'avance.
- On peut ou on doit se contenter d'accéder au dernier élément stocké.

Réciproquement :

- Si on veut pouvoir accéder à un élément quelconque à tout moment, il faudra utiliser un tableau.
- Pour cela, il est préférable de connaître au moins un majorant du nombre d'éléments à stocker.

Exercice 12.1 avec corrigé Quelle structure de données choisir pour chacune de ces tâches ?

1 Représenter un répertoire téléphonique.
2 Stocker l'historique des actions effectuées dans un logiciel et disposer d'une commande *Annuler* (ou *Undo*).
3 Comptabiliser les pièces ramassées et dépensées par un personnage dans un jeu.
4 Ranger des dossiers à traiter sur un bureau.

1 *Le nombre d'entrées du répertoire varie au cours du temps, mais on veut pouvoir accéder à n'importe quel élément. On doit donc utiliser un tableau, quitte à réserver trop de place dans celui-ci.*
2 *La commande* Annuler *n'a besoin que de connaître la dernière action effectuée. Une fois celle-ci annulée, on peut annuler l'avant-dernière, etc. Une pile est donc tout à fait appropriée.*

3 *La seule chose qui compte est la valeur totale des pièces ; leur valeur individuelle ou l'ordre dans lequel on les ramasse et dépense n'a pas d'importance. Il suffit ici d'un entier pour garder trace de la somme dont on dispose.*

4 *Si on utilise une pile, on traitera toujours en premier le dernier dossier arrivé et on risque de faire attendre longtemps les dossiers situés au bas de la pile. Pour bien faire, il faut ici tenir compte des priorités des différents dossiers, ce qui demande de les ranger dans un tableau ordonné.*

12.2 Réalisation d'une structure de pile

12.2.1 Piles à capacité finie

La manière la plus simple de réaliser une pile consiste à utiliser un tableau de taille N, avec N suffisamment grand, c'est-à-dire au moins égal au nombre maximal d'éléments qui seront stockés dans la pile. Les éléments sont rangés dans l'ordre où ils ont été empilés. Pour pouvoir empiler et dépiler, il faut connaître la position du sommet de la pile dans le tableau. Pour cela, le plus simple est de stocker le nombre d'éléments n de la pile dans la case 0 du tableau, puis les n éléments de la pile dans les cases 1 à n. On a donc la structure suivante :

0	1 … n	N
n	éléments	place disponible

	0	1	2	3	
`p = creer_pile(10)`	0				…
`empiler(p, A)`	1	A			…
`empiler(p, B)`	2	A	B		…
`empiler(p, C)`	3	A	B	C	…
`depiler(p)`	2	A	B	C	…

Les éléments colorés sont ceux qui sont réellement dans la pile. Les autres, par exemple `C` à la dernière ligne, ne sont plus accessibles et seront écrasés lorsqu'on en empilera de nouveaux.

Création d'une nouvelle pile

Pour créer une nouvelle pile de capacité `c`, on commence par allouer un tableau `p` de `c+1` cases.

```
def creer_pile(c):
    p = (c + 1) * [None]
```

On choisit d'initialiser les cases avec `None`, de manière arbitraire, ce qui n'a de toute façon aucune importance puisque le contenu initial du tableau sera écrasé lors des appels à `empiler`. Il ne reste plus qu'à stocker le nombre d'éléments (0 pour une pile vide) dans la case 0 de *p* et à renvoyer le tableau.

```
    p[0] = 0
    return p
```

Dépiler un élément

Pour dépiler le sommet d'une pile *p*, on commence par récupérer son nombre d'éléments *n* dans la première case du tableau.

```
def depiler(p):
    n = p[0]
```

On s'assure que *n* n'est pas nul, c'est-à-dire que la pile contient au moins un élément. Si ce n'est pas le cas, on fait échouer le programme.

```
    assert n > 0
```

On laisse donc au programmeur le soin de s'assurer que `taille(p)` est strictement positif avant d'appeler `depiler(p)`.

Le sommet de la pile se trouve dans `p[n]`. Avant de le renvoyer, on prend soin de décrémenter la taille de la pile.

```
    p[0] = n - 1
    return p[n]
```

Empiler un élément

Pour empiler un élément *v* dans une pile *p*, on commence par tester s'il y a de la place pour cela, sachant que la capacité de la pile est égale à `len(p)-1`.

```
def empiler(p, v):
    n = p[0]
    assert n < len(p)-1
```

Ici, on fait délibérément échouer le programme avec `assert` si la pile est pleine. Décider de ne rien faire serait une mauvaise idée : cela obligerait le programmeur à tester systématiquement la taille de la pile avant d'appeler `empiler`, au risque d'oublier de le faire et de chercher longtemps son erreur.

Si, en revanche, il y a de la place, alors on incrémente le nombre d'éléments et on stocke *v* au (nouveau) sommet de la pile.

```
    n = n + 1
    p[0] = n
    p[n] = v
```

Le code complet est donné programme 11 ci-après. Il contient également les opérations `taille`, `est_vide` et `sommet`.

PROGRAMME 11 Piles à capacité finie

Le premier élément du tableau contient le nombre d'éléments n de la pile. Les cases d'indices 1 à n du tableau contiennent alors les éléments de la pile, le sommet de la pile se trouvant à l'indice n.

```
def creer_pile(c):
    p = (c + 1) * [None]
    p[0] = 0
    return p

def depiler(p):
    n = p[0]
    assert n > 0
    p[0] = n - 1
    return p[n]

def empiler(p, v):
    n = p[0]
    assert n < len(p)-1
    n = n + 1
    p[0] = n
    p[n] = v

def taille(p):
    return p[0]

def est_vide(p):
    return taille(p) == 0

def sommet(p):
    assert taille(p) > 0
    return p[p[0]]
```

12.2.2 Piles non bornées

Un défaut de la structure de pile précédente est sa capacité bornée. En particulier, il faut être capable de déterminer une borne maximale sur le nombre d'éléments, ce qui n'est pas toujours possible.

On présente ici une seconde structure de piles, sans limite de taille. Elle exploite une propriété des tableaux de Python qu'on n'a pas encore utilisée, à savoir la possibilité d'ajouter ou de supprimer des éléments à l'extrémité droite d'un tableau en temps constant [1].

1. Il s'agit en fait de *temps constant amorti* ; voir plus loin l'encadré à ce propos.

Étant donné un tableau *p* de taille n, on peut lui ajouter un $(n+1)$-ième élément *v* à droite avec `p.append(v)`. Inversement, on peut récupérer le n-ième élément de *p* et le supprimer avec `p.pop()`, le tableau *p* prenant alors la taille $n-1$. De manière évidente, ces deux opérations correspondent exactement à `empiler(p, v)` et `depiler(p)`. Le programme 12 ci-dessous contient une réalisation de piles non bornées à l'aide de ces deux opérations.

PROGRAMME 12 Piles non bornées

Cette réalisation exploite les méthodes **append** et **pop** des tableaux de Python. On note que l'argument *c* de **creer_pile** n'est pas utilisé (mais conservé afin de garder la même interface).

```
def creer_pile(c):
    return []

def depiler(p):
    assert len(p) > 0
    return p.pop()

def empiler(p, v):
    p.append(v)

def sommet(p):
    assert len(p) > 0
    return p[-1]

def taille(p):
    return len(p)

def est_vide(p):
    return taille(p) == 0
```

POUR ALLER PLUS LOIN Tableaux redimensionnables et complexité amortie

Les tableaux de Python sont en réalité des *tableaux redimensionnables*, c'est-à-dire des tableaux dont la taille peut varier avec le temps. C'est ce qui permet notamment de fournir les opérations **append** et **pop**. Le principe d'un tableau redimensionnable est en réalité très proche de celui des piles bornées : on utilise un tableau plus grand, à l'intérieur duquel seuls certains des éléments sont significatifs. Lorsqu'il s'agit d'augmenter la taille, disons d'une unité, deux cas se présentent : soit il reste de la place et dans ce cas il n'y a rien à faire (si ce n'est se souvenir de la nouvelle taille), soit il ne reste plus de place et on alloue un nouveau tableau, deux fois plus grand, dans lequel les éléments sont recopiés et qui prend la place de l'ancien tableau. (Pour pouvoir remplacer un tableau par un autre, de manière transparente, il suffit de créer une indirection, c'est-à-dire un tableau — de taille 1 — contenant un tableau.)

Si on choisit d'allouer un nouveau tableau deux fois plus grand, et non pas seulement plus grand d'une unité, c'est pour des raisons de performances. En effet, chaque déplacement des éléments d'un tableau vers un autre a un coût proportionnel au nombre d'éléments. Allouer successivement un tableau de taille 1, puis 2... puis n aurait un coût total quadratique, alors qu'allouer un tableau de taille 1, puis 2, puis 4... puis 2^k a un coût total de l'ordre de 2^{k+1}, c'est-à-dire proportionnel à la taille finale du tableau.

➡

Dit autrement, l'ensemble des n opérations d'incrémentation de la taille n'a qu'un coût total proportionnel à n, comme si chaque opération avait eu un coût constant (même si, en réalité, certaines sont plus coûteuses que d'autres). On parle de complexité constante *amortie*.

12.3 Applications

On va maintenant présenter plusieurs programmes utilisant une pile. Ces programmes fonctionnent indifféremment avec l'une ou l'autre des réalisations présentées ci-avant.

12.3.1 Analyse des mots bien parenthésés

Comme première application des piles, on considère le problème suivant : étant donnée une chaîne de caractères ne contenant que des caractères `'('` et `')'`, déterminer s'il s'agit d'un *mot bien parenthésé*. Un mot bien parenthésé est soit le mot vide, soit la concaténation de deux mots bien parenthésés, soit un mot bien parenthésé mis entre parenthèses. Ainsi, les trois mots `''`, `'()()'` et `'(())()'` sont bien parenthésés. À l'inverse, les mots `'(()'`, `'())'` ou encore `')('` ne le sont pas. On se propose de plus d'indiquer, pour chaque parenthèse ouvrante, la position de la parenthèse fermante correspondante. Ainsi, pour le mot `'(())()'`, on donnera les couples d'indices $(0, 3)$, $(1, 2)$ et $(4, 5)$.

L'idée consiste à parcourir le mot de la gauche vers la droite et à utiliser une pile pour indiquer les indices de toutes les parenthèses ouvertes — et non encore fermées — vues jusqu'à présent. On commence donc par créer une telle pile `p` :

```
def parentheses(s):
    p = creer_pile(len(s))
```

La capacité maximale de la pile est ici la longueur du mot `len(s)` puisque, dans le pire des cas, on aura un mot composé uniquement de parenthèses ouvrantes (on rappelle qu'avec les piles non bornées, la capacité passée n'est pas significative). On parcourt alors tous les caractères du mot, de la gauche vers la droite, avec une boucle `for` :

```
    for i in range(len(s)):
```

Si le caractère est une parenthèse ouvrante, on empile son indice `i` :

```
        if s[i] == '(':
            empiler(p, i)
```

Sinon, c'est qu'il s'agit d'une parenthèse fermante[2]. Si la pile est vide, c'est que le mot n'est pas bien parenthésé, car on vient de trouver une parenthèse fermante à laquelle ne correspond aucune parenthèse ouvrante. On le signale en renvoyant immédiatement `False` :

```
        else:
            if est_vide(p):
                return False
```

Sinon, on dépile l'indice *j* de la dernière parenthèse ouvrante rencontrée et on affiche le couple `(j, i)` pour signifier que la parenthèse ouvrante à l'indice *j* correspond à la parenthèse fermante à l'indice *i* :

```
            j = depiler(p)
            print((j, i))
```

On voit ici en quoi le choix de la structure de pile est pertinent : il permet de faire correspondre chaque parenthèse fermante à la parenthèse ouvrante la plus proche, c'est-à-dire la dernière qui avait été rencontrée. Quand enfin on sort de la boucle **for**, il ne reste plus qu'à vérifier que la pile est bien vide :

```
    return est_vide(p)
```

En effet, le mot pourrait contenir plus de parenthèses ouvrantes que de parenthèses fermantes, comme `'(('`, et il faut alors signaler que le mot n'est pas bien parenthésé.

Le code complet est donné ci-après.

PROGRAMME 13 Mots bien parenthésés

```
def parentheses(s):
    p = creer_pile(len(s))
    for i in range(len(s)):
        if s[i] == '(':
            empiler(p, i)
        else:
            if est_vide(p):
                return False
            j = depiler(p)
            print((j, i))
    return est_vide(p)
```

2. On a supposé ici que le mot ne contenait que des parenthèses. Le programme pourrait être plus défensif et se prémunir contre l'éventuelle occurrence d'autres caractères.

On vérifie son résultat sur un exemple :

```
In [1]: parentheses('()(()()))')
(0, 1)
(3, 4)
(5, 6)
(2, 7)

Out[1]: True
```

Exercice 12.2 * Supposons que l'on ne souhaite pas afficher les indices des parenthèses se correspondant, mais seulement renvoyer un booléen indiquant s'il s'agit d'un mot bien parenthésé. Simplifier le programme précédent en conséquence. La structure de pile est-elle toujours nécessaire ?

Exercice 12.3 Adapter le programme 13 pour qu'il traite des mots constitués de plusieurs couples différents de symboles ouvrants et fermants, par exemple `'('` et `')'`, mais aussi `'['` et `']'` ou `'{'` et `'}'`.

Un mot est alors bien parenthésé si le symbole fermant qui correspond à chaque symbole ouvrant est du même type : le mot `'{()}[]'` est bien parenthésé mais `'[(])'` ne l'est pas.

Exercice 12.4 Adapter le programme 13 pour qu'il traite des mots constitués de parenthèses et d'autres caractères, ces derniers n'interférant pas avec les parenthèses. Ainsi le mot `'3+(4*(6-1)-2)'` est bien parenthésé, mais `'(2+6)*3)'` ne l'est pas.

Exercice 12.5 * Démontrer qu'un mot est bien parenthésé si et seulement s'il contient autant de parenthèses ouvrantes que de parenthèses fermantes et chacun de ses préfixes contient au moins autant de parenthèses ouvrantes que de parenthèses fermantes. On pourra procéder par récurrence forte sur la taille du mot.

Interpréter cette caractérisation en termes de comportement de la pile au cours de l'exécution de la fonction **parentheses**.

Exercice 12.6 Écrire une fonction qui prend un entier n en argument et renvoie le mot $(^n)^n$, c'est-à-dire le mot constitué de n parenthèses ouvrantes suivies de n parenthèses fermantes.

Exercice 12.7 Écrire une version récursive de la fonction **parentheses**. Que se passe-t-il quand on l'exécute sur le mot bien parenthésé imbriquant 1 000 paires de parenthèses (construit à l'aide de l'exercice précédent) ? La fonction **parentheses** a-t-elle ce défaut ?

12.3.2 Évaluation d'une expression arithmétique en notation polonaise inverse

Comme deuxième exemple, on souhaite réaliser un programme pour évaluer des expressions arithmétiques écrites en notation polonaise inverse (NPI), comme dans certaines calculatrices. Dans cette notation, les opérateurs arithmétiques ($+$, $*$, etc.) sont placés après leurs opérandes, en notation post-fixée. Ainsi, l'expression $2+3$ s'écrit $2\,3\,+$ et l'expression $2+3*4$ devient $2\,3\,4\ *\ +$. L'intérêt de cette notation est que les parenthèses deviennent inutiles : par exemple, l'expression $(2+3)*4$ s'écrit simplement $2\,3\ +\ 4\,*$.

Par la suite, les expressions arithmétiques en NPI sont représentées par des tableaux contenant des entiers et des caractères. Par exemple, $1\,2\ +\ 3*$ correspond au tableau `[1, 2, '+', 3, '*']`.

L'évaluation d'une expression en NPI nécessite une pile. L'idée consiste à parcourir le tableau de la gauche vers la droite et à empiler chaque nombre rencontré. Lorsque l'élément

courant est un opérateur, on dépile les deux opérandes, on effectue le calcul et on empile le résultat.

La solution que l'on propose ici fait l'hypothèse que les expressions arithmétiques contiennent uniquement les opérateurs $+$ et $*$. Il est très facile de l'étendre à d'autres opérateurs.

On commence donc par créer une pile *p* :

```
def eval_npi(exp):
    p = creer_pile(len(exp))
```

La capacité maximale de la pile est ici la longueur de l'expression arithmétique `len(exp)`, puisqu'on ne va pas empiler plus de nombres que ceux contenus dans le tableau. On parcourt alors tous les éléments du tableau `exp`, de la gauche vers la droite, avec une boucle `for` :

```
    for c in exp:
```

Si l'élément courant *c* est un opérateur arithmétique (caractère `'+'` ou `'*'`), on dépile les deux opérandes *x* et *y* de *p* et on empile `x + y` ou `x * y`, selon la valeur de *c* :

```
        if c == '+' or c == '*':
            y = depiler(p)
            x = depiler(p)
            empiler(p, x + y if c == '+' else x * y)
```

Sinon, *c* est un nombre et on l'empile dans *p* :

```
        else:
            empiler(p, c)
```

Quand on sort de la boucle `for`, il ne reste plus qu'à dépiler la valeur finale *v* de l'expression et à vérifier que la pile *p* est bien vide :

```
    v = depiler(p)
    assert est_vide(p)
    return v
```

Le code complet est donné ci-après.

PROGRAMME 14 Évaluation d'une expression en notation polonaise inverse

```
def eval_npi(exp):
    p = creer_pile(len(exp))
    for c in exp:
        if c == '+' or c == '*':
            y = depiler(p)
            x = depiler(p)
            empiler(p, x + y if c == '+' else x * y)
        else:
            empiler(p, c)
    v = depiler(p)
    assert est_vide(p)
    return v
```

On vérifie son résultat sur quelques exemples :

```
In [2]: eval_npi([1, 2, '+', 3, '*'])
Out[2]: 9
```

```
In [3]: eval_npi([1, 2, '*', 3, '+'])
Out[3]: 5
```

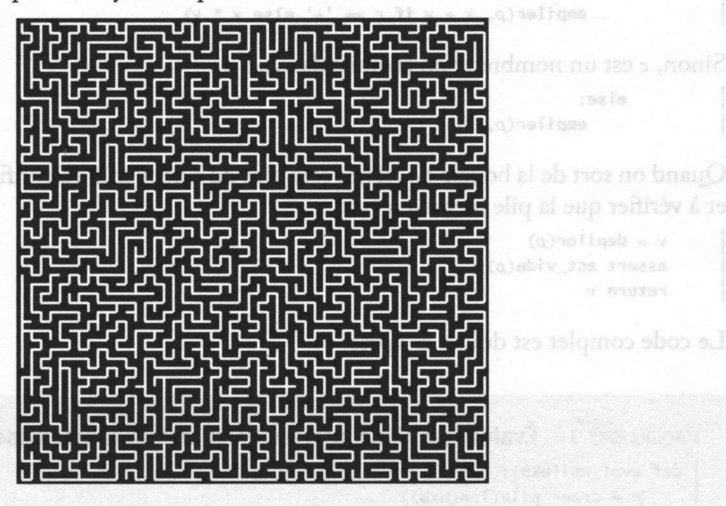
```
In [4]: eval_npi([1, 2, '+', 3, '+'])
Out[4]: 6
```

Exercice 12.8 Ajouter les opérations de soustraction et de division à la fonction **eval_npi**.

Exercice 12.9 Ajouter des opérateurs unaires, par exemple calculant la valeur absolue, le carré ou l'opposé d'un entier, à la fonction **eval_npi**.

12.3.3 Construction d'un labyrinthe parfait

On cherche ici à construire un labyrinthe parfait de dimensions données. Il s'agit d'un labyrinthe où, pour toute paire de points, il existe un et un seul chemin entre ces deux points. Voici un exemple de labyrinthe parfait de dimension 50×50 :



On peut considérer que l'entrée est en haut à gauche et la sortie en bas à droite, mais on aura compris qu'on peut tout aussi bien choisir arbitrairement deux autres points sur le bord.

On commence par se donner la dimension n du labyrinthe (en supposant qu'il soit carré, de taille $n \times n$) :

```
n = 50
```

Il faut également une matrice (n, n) de booléens indiquant, pour chaque case, si elle a déjà été atteinte par un chemin (initialement `False`) :

```
atteinte = [[False] * n for i in range(n)]
```

On se donne deux fonctions **visiter** et **est_atteinte** pour respectivement modifier et consulter le contenu de la matrice *atteinte* :

```
def visiter(c):
    (x,y) = c
    if x < 0 or x >= n or y < 0 or y >= n:
        return
    atteinte[x][y] = True

def est_atteinte(c):
    (x,y) = c
    if x < 0 or x >= n or y < 0 or y >= n:
        return True
    return atteinte[x][y]
```

La première fonction prend soin de ne pas écrire à l'extérieur de la matrice et la seconde considère les cases extérieures au labyrinthe comme déjà atteintes. Dans les deux fonctions, on commence par déconstruire l'argument *c* qui est un couple.

On écrit maintenant une fonction **choix** qui, étant donnée une position *(x,y)*, détermine les positions adjacentes non encore visitées. Le résultat est renvoyé sous la forme d'un tableau (contenant donc de 0 à 4 éléments) :

```
def choix(c):
    (x,y) = c
    r = []
    def ajouter(p):
        if not est_atteinte(p): r.append(p)
    ajouter((x-1, y))
    ajouter((x+1, y))
    ajouter((x, y-1))
    ajouter((x, y+1))
    return r
```

Le tableau résultat *r* est rempli par la fonction locale **ajouter**. Ainsi, on factorise l'appel à **est_atteinte**. On note qu'on utilise ici la méthode **append**, exactement comme on l'a fait pour écrire la fonction **empiler**.

L'étape suivante consiste en une fonction qui prend un élément au hasard dans un tableau. Elle servira à choisir aléatoirement parmi les directions possibles renvoyées par la fonction **choix** précédente :

```
def tirage(L):
    n = len(L)
    assert n > 0
    return L[random.randint(0, n-1)]
```

Cette fonction suppose que le tableau n'est pas vide (d'où le **assert**) et utilise la fonction de bibliothèque **random.randint** pour choisir un élément.

Enfin, on écrit la fonction de construction du labyrinthe, `labyrinthe`. Elle utilise une pile nommée *pile* contenant les emplacements à partir desquels on est susceptible de se déplacer. Initialement, on y place la case $(0, 0)$ et on la marque comme visitée :

```
def labyrinthe():
    pile = creer_pile(n*n)
    empiler(pile, (0,0))
    visiter((0,0))
```

Tant que cette pile n'est pas vide, on en extrait le sommet, *cellule* :

```
    while not est_vide(pile):
        cellule = depiler(pile)
```

On examine alors les déplacements encore possibles à partir de *cellule*, donnés par la fonction **choix**. S'il en existe au moins un, on en choisit un au hasard avec **tirage** :

```
        c = choix(cellule)
        if len(c) > 0:
            suivante = tirage(c)
```

On relie alors les cases *cellule* et *suivante*, par exemple en effectuant un tracé dans une fenêtre graphique. Puis on marque la case *suivante* comme étant atteinte, avec la fonction **visiter** :

```
            # c'est ici qu'on relie les cases cellule et suivante
            visiter(suivante)
```

Enfin, on remet *cellule* dans la pile, puis on ajoute *suivante*. Ainsi, le parcours reprendra à partir de *suivante* dès l'itération suivante de la boucle :

```
            empiler(pile, cellule)
            empiler(pile, suivante)
```

Une très légère optimisation consisterait à ne pas remettre *cellule* dans la pile si elle n'avait maintenant plus de voisins c'est-à-dire si *c* ne contenait qu'un seul élément. Toutefois, c'est inutilement compliqué : la prochaine fois que *cellule* sortira de la pile, on se contentera de ne rien faire.

Le code complet est donné programme 15 ci-contre.

Exercice 12.10 * Compléter le programme 15 pour effectivement créer l'image du labyrinthe (voir annexe B.2 pour la création d'une image dans un fichier).

Exercice 12.11 ** Compléter le programme 15 pour construire à la volée le chemin qui mène de l'entrée $(0, 0)$ à la sortie $(49, 49)$. Puisqu'il existe un unique chemin entre toute paire de points, il suffit pour cela de mémoriser quelle case a permis d'arriver à chaque endroit à partir de l'entrée, puis de remonter le long de ces cases à partir de la sortie.

Programme 15 Construction d'un labyrinthe parfait

```
n = 50
atteinte = [[False] * n for i in range(n)]

def visiter(c):
    (x,y) = c
    if x < 0 or x >= n or y < 0 or y >= n:
        return
    atteinte[x][y] = True

def est_atteinte(c):
    (x,y) = c
    if x < 0 or x >= n or y < 0 or y >= n:
        return True
    return atteinte[x][y]

def choix(c):
    (x,y) = c
    r = []
    def ajouter(p):
        if not est_atteinte(p): r.append(p)
    ajouter((x-1, y))
    ajouter((x+1, y))
    ajouter((x, y-1))
    ajouter((x, y+1))
    return r

def tirage(L):
    n = len(L)
    assert n > 0
    return L[random.randint(0, n-1)]

def labyrinthe():
    pile = creer_pile(n*n)
    empiler(pile, (0,0))
    visiter((0,0))
    while not est_vide(pile):
        cellule = depiler(pile)
        print(cellule)
        c = choix(cellule)
        if len(c) > 0:
            suivante = tirage(c)
            # c'est ici qu'on relie les cases cellule et suivante
            visiter(suivante)
            empiler(pile, cellule)
            empiler(pile, suivante)
```

12.4 Exercices

Dans tous les exercices proposés ici, on veillera à n'utiliser que l'interface fournie par les piles, et pas les opérateurs spécifiques aux tableaux.

Exercice 12.12 Écrire une fonction qui intervertit les deux éléments situés au sommet d'une pile de taille au moins égale à 2.

Exercice 12.13 Écrire une fonction qui dépile et renvoie le troisième élément d'une pile de taille au moins égale à 3. Les premier et deuxième éléments devront rester au sommet de la pile.

Exercice 12.14 Écrire une fonction qui lit le n-ième élément d'une pile. On s'assurera que la pile, en sortie, contient toujours les mêmes éléments. (Indication : on pourra utiliser une deuxième pile.) On prévoira le cas où la pile n'est pas de taille suffisante pour qu'un tel élément existe.

Exercice 12.15 Programmer les fonctions `sommet` et `taille` uniquement à l'aide de `empiler`, `depiler` et `est_vide`, indépendamment de la réalisation de pile choisie.

Que peut-on dire de la complexité en temps et en espace de cette fonction `taille` ?

Exercice 12.16 Écrire une fonction qui prend une pile non vide en argument et place l'élément situé à son sommet tout au fond de la pile, en conservant l'ordre des autres éléments.

Quelle est sa complexité en temps et en espace ?

Exercice 12.17 Écrire une fonction similaire à `reversed`, qui prend une pile en argument et renvoie une autre pile constituée des mêmes éléments placés dans l'ordre inverse. On s'autorise à vider la pile fournie en argument. Quelle est la complexité en temps et en espace de cette fonction ?

Exercice 12.18 * Tester dans différentes situations le comportement des boutons proposés par un navigateur Internet : *visiter une nouvelle page*, *revenir d'une page en arrière*, *aller une page en avant*.

Écrire un ensemble de fonctions simulant ces boutons. On pourra pour cet exercice utiliser deux piles.

Exercice 12.19 Écrire une fonction `couper` qui prend une pile et la coupe en enlevant de son sommet un certain nombre d'éléments (tiré au hasard) qui sont renvoyés dans une seconde pile. Exemple : si la pile initiale est `[1, 2, 3, 4, 5]` et si le nombre d'éléments retirés vaut 2, alors la pile ne contient plus que `[1, 2, 3]` et la pile renvoyée contient `[5, 4]`.

Exercice 12.20 * Mélange de cartes. Écrire une fonction `melange` qui prend en arguments deux piles et qui mélange leurs éléments dans une troisième pile de la façon suivante : tant qu'une pile au moins n'est pas vide, on retire aléatoirement un élément au sommet d'une des deux piles et on l'empile sur la pile résultat. Exemple : un mélange possible des piles `[1, 2, 3]` et `[5, 4]` est `[3, 2, 4, 1, 5]`. Note : à l'issue du mélange, les deux piles de départ sont donc vides.

Exercice 12.21 * Tour de magie de Gilbreath. Construire un paquet de cartes en empilant n fois les mêmes k cartes (par exemple, pour un paquet de 32 cartes, on empile $n = 16$ paquets de paires rouge/noir). Couper alors le paquet avec la fonction `couper` ci-dessus, puis mélanger les deux paquets obtenus à l'aide de la fonction `melange`. On observe alors que le paquet final contient toujours n blocs des mêmes k cartes (même si ces dernières peuvent apparaître dans un ordre différent au sein de chaque bloc). Sur l'exemple des 16 paquets rouge/noir, on obtient toujours 16 paquets rouge/noir ou noir/rouge.

Exercice 12.22 ** Écrire une réalisation des tableaux redimensionnables en suivant l'idée décrite dans l'encadré **Pour aller plus loin** page 305.

13

Algorithmes de tri

Nous avons montré que la recherche d'un élément dans un tableau était plus rapide si ce tableau était ordonné. Il est donc naturel de se demander s'il existe une procédure efficace pour trier des données. Nous présentons plusieurs algorithmes de tri et, surtout, nous étudions leurs complexités respectives afin de montrer qu'ils ne sont pas équivalents.

Dans tout ce chapitre, on suppose que les éléments à trier sont des entiers, mais les algorithmes présentés sont valables pour n'importe quel type d'éléments, pourvu qu'il soit muni d'un ordre total. On suppose qu'on trie des tableaux, dans l'ordre croissant. On note N le nombre d'éléments à trier.

Pour chaque tri présenté, on indique sa complexité en nombre de comparaisons et d'affectations effectuées, dans le meilleur et dans le pire des cas. La complexité en moyenne est également donnée, à titre de comparaison, mais son calcul n'est pas détaillé (la complexité en moyenne n'est pas au programme). Il est bon de savoir que la complexité optimale d'un tri effectuant uniquement des comparaisons d'éléments est en $O(N \log N)$. On en trouvera une démonstration à la fin du chapitre.

13.1 Tri par insertion

Le tri par insertion est sans doute le plus naturel. Il consiste à insérer successivement chaque élément dans l'ensemble des éléments déjà triés. C'est souvent ce que l'on fait quand on trie un jeu de cartes ou un paquet de copies.

Le tri par insertion d'un tableau `a` s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie. Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données. Il consiste à insérer successivement chaque élément `a[i]` dans la portion du tableau `a[0:i]` déjà triée. Illustrons cette idée sur un tableau de cinq entiers contenant initialement `[5,2,3,1,4]`. Au départ, `a[0:1] = 5` est déjà trié.

On insère 2 dans `a[0:1]`.	5	2	3	1	4
On insère 3 dans `a[0:2]`.	2	5	3	1	4
On insère 1 dans `a[0:3]`.	2	3	5	1	4
On insère 4 dans `a[0:4]`.	1	2	3	5	4
	1	2	3	4	5

13.1.1 Réalisation

De manière générale, chaque étape du tri par insertion correspond à la situation suivante :

0 … i-1		
… déjà trié …	`a[i]`	… à trier …

On commence par une boucle `for` pour parcourir le tableau :

```
def tri_insertion(a):
    for i, v in enumerate(a):
```

Pour insérer l'élément `a[i]` à la bonne place, on utilise alors une boucle `while` qui décale vers la droite les éléments tant qu'ils sont supérieurs à `a[i]` :

```
        j = i
        while 0 < j and v < a[j-1]:
            a[j] = a[j-1]
            j = j-1
```

Une fois sorti de la boucle, il reste à positionner `a[i]` à sa place :

```
        a[j] = v
```

Le code complet est donné programme 16 ci-dessous.

Programme 16 Tri par insertion

```
def tri_insertion(a):
    for i, v in enumerate(a):
        j = i
        while 0 < j and v < a[j-1]:
            a[j] = a[j-1]
            j = j-1
        a[j] = v
```

13.1.2 Complexité

On note que la fonction `tri_insertion` effectue exactement le même nombre de comparaisons et d'affectations. Lorsque la boucle `while` insère l'élément `a[i]` à la position $i-k$, elle effectue $k+1$ comparaisons. Au mieux, k vaut 0 et au pire, k vaut i, avec i qui varie de 1 à $N-1$, ce qui donne au final le tableau suivant :

	meilleur cas	moyenne	pire cas
comparaisons	N	$N^2/4$	$N^2/2$
affectations	N	$N^2/4$	$N^2/2$

Exercice 13.1 Dérouler à la main l'algorithme de tri par insertion sur le tableau `[15,4,2,9,55,16,0,1]`.

Exercice 13.2 * Démontrer la correction du tri par insertion.

Exercice 13.3 Proposer un exemple de tableau sur lequel le tri par insertion a un coût linéaire (meilleur cas). Proposer également un exemple de tableau sur lequel il a un coût quadratique (pire cas).

13.2 Tri rapide

Le tri rapide consiste à appliquer la méthode *diviser pour régner* : on partage les éléments à trier en deux sous-ensembles, les éléments du premier étant plus petits que les éléments du second, puis on trie récursivement chaque sous-ensemble. En pratique, on réalise le partage à l'aide d'un élément p arbitraire de l'ensemble à trier, appelé *pivot*. Les deux sous-ensembles sont alors respectivement les éléments plus petits et plus grands que p. Le tri rapide d'un tableau s'effectue en place. On va illustrer le tri rapide sur le tableau `[7,6,3,5,4,2,1]`.

Pour trier `a[0:7]`, on choisit au hasard 4 comme pivot.	7	6	3	5	4	2	1
On place les éléments plus petits que 4, puis 4, puis les autres.	3	2	1	**4**	7	6	5
Pour trier `a[0:3]`, on choisit 3 comme pivot.	3	2	1				
On place les éléments plus petits que 3, puis 3, puis les autres.	2	1	**3**				
Pour trier `a[0:2]`, on choisit 2 comme pivot.	2	1					
On place les éléments plus petits que 2, puis 2, puis les autres.	1	**2**					
Pour trier `a[4:7]`, on choisit au hasard 6 comme pivot.					7	6	5
On place les éléments plus petits que 6, puis 6, puis les autres.					5	**6**	7
On obtient finalement :	1	2	3	4	5	6	7

13.2.1 Réalisation

Pour réaliser ce tri, on écrit deux fonctions. Une fonction de partition organise les éléments autour d'un pivot et renvoie la position de ce dernier. Une autre fonction trie récursivement les deux portions du tableau à gauche et à droite du pivot. Les fonctions de partition et de tri prennent en arguments le tableau et deux indices délimitant la portion à considérer.

On commence par écrire une fonction **echange** pour échanger les éléments `a[i]` et `a[j]` d'un tableau `a` :

```
def echange(a, i, j):
    a[i], a[j] = a[j], a[i]
```

La fonction **partition** prend le tableau `a` et deux indices `g` et `d` en arguments, avec la convention que `g` est inclus et `d` exclu. On suppose qu'il y a au moins un élément dans ce segment, ce que l'on vérifie avec **assert** :

```
def partition(a, g, d):
    assert g < d
```

On choisit `a[g]` comme pivot :

```
    v = a[g]
```

Le principe consiste alors à parcourir le tableau de la gauche vers la droite, entre les indices `g` (inclus) et `d` (exclu), avec une boucle **for**. À chaque itération, la situation est la suivante :

g   m   i   d

v	< v	⩾ v	?

L'indice `i` de la boucle dénote le prochain élément à considérer et l'indice `m` partitionne la portion déjà parcourue.

```
    m = g
    for i in range(g+1, d):
```

Si `a[i]` est supérieur ou égal à `v`, il n'y a rien à faire. Dans le cas contraire, pour conserver l'invariant de boucle, il suffit d'incrémenter `m` et d'échanger `a[i]` et `a[m]` :

```
        if a[i] < v:
            m = m+1
            echange(a, i, m)
```

Une fois sorti de la boucle, on met le pivot à sa place, c'est-à-dire à la position `m`, et on renvoie cet indice :

```
    if m != g:
        echange(a, g, m)
    return m
```

On écrit ensuite la partie récursive du tri rapide sous la forme d'une fonction **tri_rapide_rec** qui prend les mêmes arguments que la fonction **partition**. Si $g \geqslant d - 1$, il y a au plus un élément à trier et il n'y a donc rien à faire, ce qui assure au passage que l'on n'appelle pas **partition** avec $g \geqslant d$:

```
def tri_rapide_rec(a, g, d):
    if g >= d-1: return
```

Sinon, on partitionne les éléments entre `g` et `d` :

```
    m = partition(a, g, d)
```

Après cet appel, le pivot `a[m]` se retrouve à sa place définitive. On effectue alors deux appels récursifs pour trier `a[g..m[` et `a[m+1..d[` :

```
    tri_rapide_rec(a, g, m)
    tri_rapide_rec(a, m+1, d)
```

Pour trier un tableau, il suffit d'appeler **tri_rapide_rec** sur la totalité de ses éléments :

```
def tri_rapide(a):
    tri_rapide_rec(a, 0, len(a))
```

Tel qu'il est écrit, ce code présente deux inconvénients. D'une part, il atteint très vite le nombre maximal d'appels récursifs (1 000 par défaut en Python). D'autre part, il peut exhiber une complexité quadratique, notamment dans le cas d'un tableau déjà trié. On va remédier à ces deux problèmes.

Pour rester sous la limite des 1 000 appels récursifs de Python, on va tout d'abord supprimer l'un des deux appels récursifs au profit d'une boucle.

```
def tri_rapide_rec(a, g, d):
    while g < d-1:
        m = partition(a, g, d)
        tri_rapide_rec(a, g, m)
        g = m+1
```

Ici, le second appel a été remplacé par l'affectation `g = m+1` et le tout a été placé dans une boucle **while**, pour que le calcul soit répété tant que le segment contient au moins deux éléments. Cela ne suffit pas pour autant, car l'appel récursif restant peut être répété un grand nombre de fois, par exemple si le pivot se trouve plus souvent dans la moitié droite que gauche. D'où l'idée d'utiliser l'appel récursif pour le plus petit segment et la boucle pour le plus grand :

```
        if m-g < d-m-1:
            tri_rapide_rec(a, g, m)
            g = m+1
        else:
            tri_rapide_rec(a, m+1, d)
            d = m
```

Pour ce qui est de la complexité quadratique dans le cas d'un tableau déjà trié (ou trié en ordre inverse par exemple), une solution simple consiste à ne pas choisir systématiquement le premier élément du segment comme pivot, mais plutôt un élément au hasard. Une façon très simple de réaliser cette idée consiste à démarrer la fonction **partition** par un échange aléatoire :

```
def partition(a, g, d):
    echange(a, g, random.randint(g, d-1))
    ...
```

Le reste du code est alors inchangé. Le code complet est donné ci-après.

PROGRAMME 17 Tri rapide

```
def echange(a, i, j):
    a[i], a[j] = a[j], a[i]

def partition(a, g, d):
    assert g < d
    echange(a, g, random.randint(g, d-1))
    v = a[g]
    m = g
    for i in range(g+1, d):
        if a[i] < v:
            m = m+1
            echange(a, i, m)
    if m != g:
        echange(a, g, m)
    return m

def tri_rapide_rec(a, g, d):
    while g < d-1:
        m = partition(a, g, d)
        if m-g < d-m-1:
            tri_rapide_rec(a, g, m)
            g = m+1
        else:
            tri_rapide_rec(a, m+1, d)
            d = m

def tri_rapide(a):
    tri_rapide_rec(a, 0, len(a))
```

13.2.2 Complexité

La fonction `partition` fait toujours exactement $d - g - 1$ comparaisons. Si la fonction `partition` détermine un segment de longueur K et un autre de longueur $N-1-K$, la fonction `tri_rapide_rec` va donc effectuer $N - 1$ comparaisons par l'intermédiaire de `partition`, puis d'autres comparaisons par l'intermédiaire des deux appels récursifs à `tri_rapide_rec` (la fonction réécrite avec un seul appel récursif a la même complexité en temps). Le pire des cas correspond à $K = 0$, ce qui donne, en notant $C(N)$ la complexité du tri d'un tableau de longueur N, l'équation de récurrence suivante :

$$C(N) = N - 1 + C(N - 1),$$

d'où $C(N) \sim \frac{N^2}{2}$. Le meilleur des cas correspond à un segment coupé en deux moitiés égales, c'est-à-dire $K = N/2$. L'équation de récurrence devient alors :

$$C(N) = N - 1 + 2C(N/2).$$

On en déduit facilement $C(N) \sim N \log N$.

En ce qui concerne le nombre d'affectations, on note que la fonction `partition` effectue un appel à `echange` initial, autant d'appels à `echange` que d'incrémentations de `m`, et éventuellement un dernier appel lorsque `m != g`. Le meilleur des cas est atteint lorsque le pivot est toujours à sa place. Il y a alors un seul appel à `echange`, soit deux affectations. Il est important de noter que ce cas ne correspond pas à la meilleure complexité en termes de comparaisons (qui est alors quadratique). Dans le pire des cas, le pivot se retrouve toujours à la position `r-1`. La fonction `partition` effectue alors $2(d-g)$ affectations, d'où un total de N^2 affectations.

	meilleur cas	moyenne	pire cas
comparaisons	$N \log N$	$2N \log N$	$N^2/2$
affectations	$2N$	$2N \log N$	N^2

Exercice 13.4 Dérouler à la main l'algorithme de tri rapide sur le tableau `[15,4,2,8,17,23,0,1]`.

Exercice 13.5 * Proposer un exemple de tableau sur lequel le tri rapide a un coût en $O(N \log N)$ (meilleur cas). Proposer également un exemple de tableau sur lequel il a un coût quadratique (pire cas).

13.3 Tri fusion

Comme le tri rapide, le tri fusion applique le principe *diviser pour régner*. Il partage les éléments à trier en deux parties de même taille, sans chercher à comparer leurs éléments. Une fois les deux parties triées récursivement, il les fusionne, d'où le nom de tri fusion. Ainsi on évite le pire cas du tri rapide où les deux parties sont de tailles disproportionnées.

• Pour trier `a[0:8]`, on trie `a[0:4]` et `a[4:8]`.	7	6	3	5	4	2	1	8
• Pour trier `a[0:4]`, on trie `a[0:2]` et `a[2:4]`.	7	6	3	5				
• Pour trier `a[0:2]`, on trie `a[0:1]` et `a[1:2]`.	7	6						
• On fusionne `a[0:1]` et `a[1:2]`.	6	7						
• Pour trier `a[2:4]`, on trie `a[2:3]` et `a[3:4]`.			3	5				
• On fusionne `a[2:3]` et `a[3:4]`.			3	5				
• On fusionne `a[0:2]` et `a[2:4]`.	3	5	6	7				
• Pour trier `a[4:8]`, on trie `a[4:6]` et `a[6:8]`.					4	2	1	8

• Pour trier `a[4:6]`, on trie `a[4:5]` et `a[5:6]`.					4	2		
• On fusionne `a[4:5]` et `a[5:6]`.					2	4		
• Pour trier `a[6:8]`, on trie `a[6:7]` et `a[7:8]`.							1	8
• On fusionne `a[6:7]` et `a[7:8]`.							1	8
• On fusionne `a[4:6]` et `a[6:8]`.					1	2	4	8
• On fusionne `a[0:4]` et `a[4:8]`.	1	2	3	4	5	6	7	8

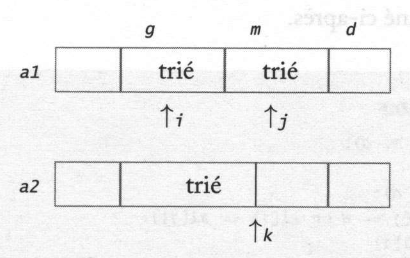
13.3.1 Réalisation

On va chercher à réaliser le tri fusion d'un tableau en place, en délimitant la portion à trier par deux indices `g` (inclus) et `d` (exclu). Pour le partage, il suffit de calculer l'indice médian $m = \frac{g+d}{2}$. On trie alors récursivement les deux parties délimitées par `g` et `m` d'une part, `m` et `d` d'autre part. Il reste à effectuer la fusion. Il s'avère extrêmement difficile de la réaliser en place. Le plus simple est d'utiliser un second tableau, alloué une et une seule fois au début du tri.

On commence par écrire la fonction **`fusion`**. Elle prend en arguments deux tableaux, `a1` et `a2`, et les trois indices `g`, `m` et `d`. Les portions `a1[g..m[` et `a1[m..d[` sont supposées triées. L'objectif est de les fusionner dans `a2[g..d[`. Pour cela, on va parcourir les deux portions de `a1` avec deux variables `i` et `j` et la portion de `a2` à remplir avec une boucle **`for`** :

```
def fusion(a1, a2, g, m, d):
    i, j = g, m
    for k in range(g, d):
```

À chaque itération, la situation est donc la suivante :



Il faut alors déterminer la prochaine valeur à placer en `a2[k]`. Il s'agit de la plus petite des deux valeurs `a1[i]` et `a1[j]`. Il convient cependant de traiter correctement le cas où il n'y a plus d'élément dans l'une des deux moitiés. On détermine si l'élément doit être pris dans la moitié gauche avec le test suivant :

```
        if i < m and (j == d or a1[i] <= a1[j]):
```

Dans les deux cas, on copie l'élément dans *a2[k]* et on incrémente l'indice correspondant.

```
            a2[k] = a1[i]
            i = i+1
        else:
            a2[k] = a1[j]
            j = j+1
```

On écrit ensuite la fonction **tri_fusion**. On commence par allouer un tableau temporaire *tmp* en faisant une copie du tableau à trier :

```
def tri_fusion(a):
    tmp = a[:]
```

La partie récursive du tri fusion est matérialisée par une fonction récursive locale **tri_fusion_rec** qui prend en arguments les indices *g* et *d* délimitant la portion à trier :

```
    def tri_fusion_rec(g, d):
```

Si le segment contient au plus un élément, c'est-à-dire si $g \geqslant d-1$, il n'y a rien à faire :

```
        if g >= d-1: return
```

Sinon, on partage l'intervalle en deux moitiés égales : on calcule l'élément médian *m*, puis on trie récursivement *a[g..m[* et *a[m..d[* :

```
        m = (g+d)//2
        tri_fusion_rec(g, m)
        tri_fusion_rec(m, d)
```

Il reste à effectuer la fusion. Pour cela, on copie toute la portion *a[g..d[* dans le tableau *tmp*, puis on appelle la fonction **fusion**, qui fusionne le tout dans *a* :

```
        tmp[g:d] = a[g:d]
        fusion(tmp, a, g, m, d)
```

Enfin, on trie le tableau *a* tout entier en appelant **tri_fusion_rec** sur la totalité de ses éléments :

```
    tri_fusion_rec(0, len(a))
```

Le code complet est donné ci-après.

Programme 18 Tri fusion

```
def fusion(a1, a2, g, m, d):
    i, j = g, m
    for k in range(g, d):
        if i < m and (j == d or a1[i] <= a1[j]):
            a2[k] = a1[i]
            i = i+1
        else:
            a2[k] = a1[j]
            j = j+1
```

```
def tri_fusion(a):
    tmp = a[:]
    def tri_fusion_rec(g, d):
        if g >= d-1: return
        m = (g+d)//2
        tri_fusion_rec(g, m)
        tri_fusion_rec(m, d)
        tmp[g:d] = a[g:d]
        fusion(tmp, a, g, m, d)
    tri_fusion_rec(0, len(a))
```

13.3.2 Complexité

Si on note $C(N)$ (resp. $f(N)$) le nombre total de comparaisons effectuées par **tri_fusion** (resp. **fusion**) pour trier un tableau de longueur N, on a l'équation de récurrence suivante :

$$C(N) = 2C(N/2) + f(N)$$

En effet, les deux appels récursifs se font sur deux segments de même longueur $N/2$. Dans le meilleur des cas, la fonction **fusion** n'examine que les éléments de l'un des deux segments car ils sont tous plus petits que ceux de l'autre segment. Dans ce cas, $f(N) = N/2$ et donc $C(N) \sim \frac{1}{2}N \log N$. Dans le pire des cas, tous les éléments sont examinés par **fusion** et donc $f(N) = N - 1$, d'où $C(N) \sim N \log N$.

Le nombre d'affectations est le même dans tous les cas : N affectations dans la fonction **fusion** (chaque élément est copié de *a1* vers *a2*) et N affectations effectuées par la copie de *a* vers *tmp*. Si on note $A(N)$ le nombre total d'affectations pour trier un tableau de longueur N, on a donc :

$$A(N) = 2A(N/2) + 2N,$$

d'où un total de $2N \log N$ affectations.

	meilleur cas	moyenne	pire cas
comparaisons	$\frac{1}{2}N \log N$	$N \log N$	$N \log N$
affectations	$2N \log N$	$2N \log N$	$2N \log N$

On note que, dans tous les cas, la complexité du tri fusion est la même. Cette complexité est optimale.

SAVOIR-FAIRE Distinguer par leurs complexités deux algorithmes résolvant un même problème

Il convient tout d'abord de s'assurer que les algorithmes résolvent bien le même problème : il n'est pas rare que leurs conditions d'utilisation soient différentes, ce qui rend moins pertinente une comparaison de complexité.

- On s'intéressera d'abord à la complexité en temps dans le pire des cas, qui est souvent la plus représentative.
- On discutera cependant si ce pire cas a des chances de se présenter dans des situations réelles.
- On n'oubliera pas d'étudier la complexité en espace, qui peut départager des algorithmes de performances par ailleurs similaires.

Exercice 13.6 avec corrigé Quelles sont les différences entre les algorithmes de tri rapide et de tri fusion du point de vue de la complexité en temps et en espace ?

Quelles conséquences cela a-t-il pour leur utilisation ?

On rappelle tout d'abord que ces deux tris fonctionnent sur les mêmes entrées sans conditions particulières.

- *Le tri rapide a une complexité en temps quadratique dans le pire cas. Si on veut être certain que ce cas ne se présente pas, on choisira plutôt le tri fusion, qui est au pire en $O(n \log n)$.*
- *Ces deux tris ont une complexité moyenne en $O(n \log n)$, ce qui signifie qu'en général ils auront des performances comparables, en particulier si la répartition des données dans le tableau à trier n'est pas trop particulière.*
- *Le tri fusion a une complexité en espace légèrement supérieure à celle du tri rapide puisque la fusion ne s'opère pas en place. Dans un cas où la mémoire est une ressource critique, on évitera donc de choisir le tri fusion.*

POUR ALLER PLUS LOIN La complexité optimale du tri

La meilleure complexité que l'on peut espérer d'un tri effectuant uniquement des comparaisons d'éléments est en $O(N \log N)$. En effet, on peut visualiser un tel algorithme comme un arbre binaire. Chaque nœud interne représente une comparaison effectuée, le sous-arbre gauche (resp. droit) représentant la suite de l'algorithme lorsque le test est positif (resp. négatif). Chaque feuille représente un résultat possible, c'est-à-dire une permutation effectuée sur la séquence initiale. Si on suppose les N éléments distincts, il y a $N!$ permutations possibles, donc au moins $N!$ feuilles à cet arbre. Sa hauteur est donc au moins égale à $\log N!$. Or le plus long chemin de la racine à une feuille représente le plus grand nombre de comparaisons effectuées par l'algorithme sur une entrée. Il existe donc une entrée pour laquelle le nombre de comparaisons est au moins $\log N!$. Par la formule de Stirling, on sait que $\log N! \sim N \log N$. Pour une démonstration plus détaillée, on pourra consulter *The Art of Computer Programming* [Vol 3, Sec. 5.3].

Exercice 13.7 Dérouler à la main l'algorithme de tri fusion sur le tableau `[5,40,2,18,17,3,0,1,14]`.

13.4 Exercices

Exercice 13.8 ** Le tri par sélection consiste, comme le tri par insertion, à maintenir à chaque itération i une portion du tableau `a[0:i]` déjà triée. En revanche, au lieu de chercher à insérer `a[i]` dans cette portion, on recherche le minimum des éléments de `a[i:n]` et on échange ce minimum avec `a[i]`. La portion du tableau `a[0:i+1]` est alors triée.

1 Écrire une fonction qui réalise le tri par sélection.
2 Démontrer la correction de ce tri.
3 Évaluer sa complexité en temps, en espace, dans le meilleur et dans le pire des cas.

Exercice 13.9 * Montrer que, dans le cas du tri rapide du programme 17 page 321, le nombre maximal d'appels imbriqués à la fonction **tri_rapide_rec** ne peut excéder $\log N$.

Exercice 13.10 ** Étant donné un entier k, on dit qu'un tableau est *k-presque trié* :

- si chacun de ses éléments est à au plus k indices de la position où il devrait être ;
- ou bien si au plus k de ses éléments ne sont pas à leur place.

Démontrer qu'à k fixé, le tri par insertion a une complexité en $O(n)$ sur les tableaux k-presque triés.

Exercice 13.11 * Une idée classique pour accélérer un algorithme de tri consiste à effectuer un tri par insertion quand le nombre d'éléments à trier est petit, c'est-à-dire devient inférieur à une constante fixée à l'avance (par exemple 5). Modifier le tri rapide de tableaux pour prendre en compte cette idée. On pourra reprendre la fonction **tri_insertion** et la généraliser en lui passant deux indices *g* et *d* pour délimiter la portion du tableau à trier.

Exercice 13.12 ** **Calcul rapide de la médiane.** L'algorithme que l'on écrit ici permet de déterminer la médiane, et même plus généralement le k-ième élément d'un tableau, sans le trier intégralement. Il est linéaire dans le pire des cas.

1 Écrire une fonction qui prend en argument un tableau de cinq éléments et calcule sa médiane.
2 Écrire une fonction qui prend en argument un tableau quelconque, le divise en groupes de cinq éléments et construit le tableau des médianes de chaque groupe de cinq.
3 Modifier la fonction précédente pour qu'elle s'appelle récursivement sur le « tableau des médianes » construit.
4 Enfin, écrire une fonction qui effectue une partition du tableau de départ avec pour pivot la « médiane des médianes » calculée précédemment.
Pour trouver le k-ième élément du tableau, où doit-on le chercher en fonction des tailles des deux sous-tableaux délimités par la partition ? Programmer l'appel récursif correspondant.
5 Pourquoi n'est-ce pas une bonne idée d'extraire les groupes de cinq éléments avec la construction `t[i:i+5]` ? Comment peut-on procéder autrement ?
6 De même, on pourra chercher une façon de construire en place le tableau des médianes.

Exercice 13.13 * Une façon d'optimiser la fonction **tri_fusion** consiste à éviter la fusion lorsque, à l'issue des deux appels récursifs, les éléments de la moitié gauche se trouvent être tous plus petits que les éléments de la moitié droite. On le teste facilement en comparant l'élément le plus à droite de la moitié gauche et l'élément le plus à gauche de la moitié droite. Modifier la fonction **tri_fusion** en suivant cette idée.

Exercice 13.14 ** Pour éviter la copie de *a* vers *tmp* (avec `tmp[g :d] = a[g :d]`) dans la fonction **tri_fusion**, une idée consiste à trier les deux moitiés du tableau *a* tout en les déplaçant vers le tableau *tmp*, puis à fusionner de *tmp* vers *a* comme on le fait déjà. Cependant, pour trier les éléments de *a* vers *tmp*, il faut, inversement, trier les deux moitiés en place puis fusionner vers *tmp*. On a donc besoin de deux fonctions de tri mutuellement récursives. On peut cependant n'en n'écrire qu'une seule, en passant un paramètre supplémentaire indiquant si le tri doit être fait en place ou vers *tmp*. Modifier les fonctions **tri_fusion** et **tri_fusion_rec** en suivant cette idée.

Exercice 13.15 * Comme pour le tri rapide, on peut terminer le tri fusion par un tri par insertion lorsque le nombre d'éléments à trier devient petit.

Annexe

Travaux pratiques et compléments

Cette annexe doit amener l'étudiant à goûter les implications concrètes des différentes parties du cours, d'abord à travers des travaux pratiques — ludiques pour certains, allant de la dissection d'un ordinateur pour en manipuler les composants matériels, à l'écriture de procédures cryptographiques que l'on utilise quotidiennement parfois sans même en avoir conscience, en passant par des algorithmes de création et de manipulation d'images.
La dernière annexe sur les entrées/sorties précise comment écrire dans des fichiers, des images, et produire des tracés géométriques.

A
Travaux pratiques

A.1 Création de programmes autonomes

A.1.1 Compilation d'un programme

Dans cette première partie du TP, on verra comment créer un programme autonome. Pour cela, on l'écrira dans un langage informatique compréhensible par les humains et on le transformera en un code exécutable grâce à un compilateur. Les systèmes Unix proposent en standard un compilateur d'un langage appelé C, ce qui n'est pas le cas des systèmes Microsoft. L'installation d'un tel outil sous MS-Windows sortant du cadre de cet ouvrage, on se contentera de décrire ce qu'il convient de faire sous un système Unix [1]. L'étude du langage C étant hors du cadre de cet ouvrage, on se contentera d'un programme très simple.

Il convient tout d'abord de créer, en utilisant la fenêtre d'édition de Spyder, un fichier `hello.c` contenant le texte du programme :

```
#include <stdio.h>

int main() {
  printf("Hello\n");
}
```

1. Il peut arriver que le compilateur ne soit pas installé par défaut sous votre système. Cependant sous tout système GNU/Linux, il suffira d'installer le paquet des outils de développement pour que le compilateur C soit installé. Sous toute distribution dérivée de la Debian, en particulier sous Ubuntu, installer le paquet `build-essential` est suffisant.

Enregistrer ce fichier. On supposera dans la suite qu'il est enregistré dans le répertoire `/home/jdupont/TP`. Il conviendra de remplacer ce nom par le nom du répertoire où ce fichier a été enregistré.

Lancer alors un émulateur de terminal et taper la commande suivante, suivie de la touche *Entrée* :

```
cc -o /home/jdupont/TP/hello /home/jdupont/TP/hello.c
```

Cette commande lance le compilateur C sur `hello.c` et construit le programme exécutable `hello` dans le même répertoire. Constater sa présence à partir de l'explorateur de fichiers et s'assurer que le compilateur C a bien rendu ce fichier exécutable.

On peut alors exécuter ce fichier dans le terminal en tapant :

```
/home/jdupont/TP/hello
```

Le programme s'exécute alors et devrait vous gratifier d'un laconique *Hello*.

Le programme exécutable est un fichier écrit en langage machine, donc incompréhensible pour les humains. Cependant, il est possible d'en donner une représentation plus compréhensible (on dit qu'on *désassemble* le programme). Pour cela, taper la commande suivante :

```
objdump -d /home/jdupont/TP/hello | more
```

Devrait alors s'afficher quelque chose dont le début ressemble à ceci (si votre système est un PC dont le système est en 64 bits) :

```
/home/jdupont/TP/hello :     file format elf64-x86-64


 Disassembly of section .init :

 00000000004003b8 <_init> :
   4003b8 : 48 83 ec 08          sub    $0x8,%rsp
   4003bc : e8 6b 00 00 00       callq  40042c <call_gmon_start>
   4003c1 : 48 83 c4 08          add    $0x8,%rsp
   4003c5 : c3                   retq
```

On peut faire défiler tout le programme désassemblé avec la barre d'espace.

Sur les quatre dernières lignes présentées ici, la première colonne est une liste d'adresses mémoire, écrites en hexadécimal (voir exercice 2.7) . Il s'agit des adresses où seront chargées les instructions du programme. La seconde colonne en montre le contenu, écrit également sous forme hexadécimale. Enfin, les dernières colonnes donnent une représentation sous forme plus compréhensible des instructions que représentent ces données binaires. À la première ligne, on trouve par exemple l'instruction `sub` qui soustrait ici la valeur 8 du registre nommé `%rsp`, puis un branchement à l'adresse `40042c` (instruction `call`), puis une instruction add, ajoutant la valeur 8 au registre `%rsp`.

A.1.2 Exécution autonome d'un programme Python

On explique à présent comment un programme créé avec Spyder peut être exécuté de façon autonome, ou plus précisément sans Spyder.

Créer à l'aide de Spyder un fichier `hellotk.py` avec le contenu suivant :

```
#! /usr/bin/python3

from tkinter import *
w = Tk()
b = Button(w, text='Quitter', command=w.quit)
b.pack()
mainloop()
```

Enregistrer ce fichier et quitter Spyder. On supposera qu'il est placé dans le répertoire `/home/jdupont/TP` sous un système Unix, ou `U:\Jean Dupont\TP` sous un système MS-Windows.

Ce petit programme lance une interface graphique dotée d'une seule fonctionnalité : un bouton *Quitter* (voir figure A.1).

Quel que soit votre système, une première façon de le lancer est d'ouvrir un terminal et de taper la commande

```
python /home/jdupont/TP/hellotk.py
```

ou, sous Windows :

```
python "U:\Jean Dupont\TP\hellotk.py"
```

Il existe cependant une deuxième méthode plus simple, à condition d'effectuer une petite manipulation préalable :

- **Sous MS-Windows et Mac OS X.** Renommer le fichier en `hellotk.pi` suffit pour indiquer au système que le fichier ne doit pas être ouvert avec Spyder, mais directement exécuté par l'interpréteur Python.
- **Sous un système de type Unix.** Il suffit de donner la permission d'exécuter le fichier. Les deux premiers caractères du fichier (`#!`) indiquent que ce qui suit est le nom du programme à lancer pour exécuter le fichier[2].



Figure A.1
Interface graphique lancée par le programme `hellotk.py`

2. Sous Unix, Python est en général installé dans `/usr/bin/python`, mais il peut être nécessaire d'adapter cette ligne suivant votre installation. Si `/usr/bin/python` ne fonctionne pas, essayer avec `/usr/bin/env python` qui demande au système d'exécuter la commande `env`, laquelle se chargera de chercher elle-même la commande `python` et devrait la trouver si votre système est configuré raisonnablement.

Avec cette modification, ce script peut désormais être exécuté par simple (ou double) clic dans l'explorateur de fichiers du système. On peut aussi l'envoyer à une autre personne, qui pourra l'exécuter sans avoir à regarder son contenu, ni même à installer l'environnement de développement Spyder, dès lors que Python est installé sur sa machine. C'est le cas par défaut sur tous les systèmes GNU/Linux ainsi que sous Mac OS X, mais pas sur les systèmes MS-Windows. Pour ces derniers, l'extension de Python `py2exe` permet, à partir d'un programme, de produire un exécutable qui tourne même sans installation de Python.

A.2 Mémoire virtuelle et performances de l'ordinateur

La mémoire vive d'un ordinateur est limitée. Le système d'exploitation, mais aussi chaque application ouverte sur un ordinateur consomme de la mémoire vive et il arrive bien souvent que celle-ci soit insuffisante pour ouvrir simultanément autant d'applications qu'on le désire.

Pour pallier ce problème, les concepteurs de systèmes d'exploitation ont inventé la *mémoire virtuelle*. C'est un mécanisme par lequel le système fait croire aux applications qu'elles disposent d'une plus grande mémoire vive qu'il n'en existe physiquement.

Pour avoir une idée de ce que fait alors le système, on imagine qu'un secrétaire se met à son bureau pour étudier des dossiers. Son plan de travail est encombré d'une montagne de papiers (d'autres dossiers en cours), mais heureusement il dispose d'un elfe de maison, à qui il a donné l'ordre de libérer une place suffisante sur son bureau pour travailler. C'est pourquoi, dès que le secrétaire a besoin d'un peu de place, l'elfe prend une partie des dossiers encombrant le bureau et va les ranger dans une armoire pour libérer de la place. Dès que le secrétaire s'apprête à travailler de nouveau sur un dossier que l'elfe a rangé, celui-ci lit dans ses pensées et remet le dossier sur le bureau, après avoir au besoin rangé un autre dossier dans l'armoire pour faire de la place si nécessaire.
Si de plus, l'elfe hypnotise le secrétaire, d'une part pour le faire patienter le temps qu'il effectue ses manipulations et d'autre part pour qu'il croie être à chaque fois en train de travailler sur une portion de bureau différente, il ne s'apercevra de rien.

C'est exactement ce qui se passe dans un ordinateur, avec le système d'exploitation dans le rôle de l'elfe, les applications dans celui du secrétaire, la mémoire vive dans le rôle du bureau et le disque dur dans celui de l'armoire : lorsqu'une application réclame plus de mémoire qu'il n'en existe physiquement, le système d'exploitation enregistre une portion de mémoire vive sur une partie du disque dur réservée à cet effet et la donne à l'application pour qu'elle puisse travailler.

Si la portion de mémoire enregistrée est de nouveau requise, le système d'exploitation la recharge en mémoire vive, après avoir au besoin enregistré une autre portion de mémoire sur le disque pour faire de la place (il y a donc un ballet d'échanges de données entre disque dur et mémoire vive ; la partie du disque dur est pour cela appelée espace ou *fichier d'échange*, ou *swap* en anglais). Du point de vue des applications, comme de celui de l'utilisateur, tout se passe comme s'il y avait davantage de mémoire vive qu'il n'y en a matériellement...

Ou presque. Il y a en effet une différence notable : le temps d'accès à des données sur disque est beaucoup plus important que le temps d'accès à des données en mémoire vive (d'un facteur en général supérieur à 10^3). Lorsque la quantité de mémoire virtuelle utilisée est faible, les données ou applications transférées sur le disque sont celles dont l'utilisateur se sert le moins à ce moment, mais si elle devient plus importante, des données plus fréquemment utilisées sont transférées sur disque et doivent y être fréquemment relues. Pendant ces relectures, l'application concernée est contrainte d'attendre. Les performances globales de l'ordinateur s'en ressentent fatalement.

EN PRATIQUE Ça rame...

Tout le monde a déjà eu affaire à un ordinateur qui semblait fonctionner au ralenti, où la moindre ouverture ou minimisation d'une fenêtre demandait un temps considérable. La cause en est souvent la même : un manque de RAM par rapport à ce que demandent le système d'exploitation et les applications[3]. Le système d'exploitation doit alors sans cesse échanger des données entre mémoire vive et disque dur pour répondre aux sollicitations des applications.

Pour résoudre le problème, il n'y a que deux possibilités :

- Diminuer l'usage de la mémoire vive. Pour cela, il convient de fermer les applications inutilisées, d'utiliser des applications moins gourmandes, voire de changer de système d'exploitation (certaines distributions GNU/Linux comme Toutou Linux se contentent de 128 Mo de mémoire vive, là où d'autres systèmes réclament au minimum 1 Go).
- Lorsque c'est possible, augmenter la mémoire vive en ouvrant l'ordinateur pour ajouter des barrettes de mémoire ou remplacer les barrettes en place par d'autres de plus grande capacité. En pratique, c'est difficile à faire au bout de quelques années, car d'une part les formats de barrettes de mémoire vive changent souvent et d'autre part, il y a des limitations à la quantité de mémoire vive que la carte mère d'un PC peut accepter. Il convient donc d'être attentif au moment de l'achat d'un PC : les étiquettes vantent surtout la fréquence du processeur, il vaut mieux s'intéresser d'abord à la quantité de mémoire vive.

3. Cela arrive typiquement avec un ordinateur ancien sur lequel on veut mettre un système d'exploitation ou des applications récentes.

Ce TP consiste à lancer un programme créant n mots de données dans la mémoire (vive ou virtuelle), puis accédant au hasard à N endroits dans les données créées. Si les données sont toutes dans la mémoire vive, l'accès est rapide. Si elles sont en quantité tellement importante qu'une partie a dû être stockée sur le disque dur, l'accès est plus lent.

1 Télécharger le programme `mem_virtuelle.py` sur le site de cet ouvrage [4] et l'ouvrir dans un IDE Python. L'entier N y est fixé à 1000 et n peut être arbitrairement choisi.

2 Faire exécuter le code par l'IDE ; cela ne produit en principe aucun résultat. Ensuite, dans le même interpréteur interactif, lancer la fonction `teste` avec pour argument l'entier n.

3 Si on lance par exemple `teste(100000000)`, combien de mots mémoire seront-ils créés ? Combien de Mo cela représente-t-il sur des systèmes respectivement 32 et 64 bits ?

4 La fonction `teste` renvoie alors le temps, en secondes, nécessaire pour accéder à N de ces mots mémoire au hasard parmi les n. Elle effectue en fait plusieurs fois le test avant de rendre la valeur la plus faible (qui est la plus représentative de ce qui se passe dans le meilleur des cas).

   Tester différentes valeurs de n et essayer de déterminer comment évolue la valeur renvoyée par le programme en fonction de n. Pour n trop grand, le programme s'arrêtera brutalement car il n'y aura tout simplement pas assez de mémoire totale (même avec la mémoire virtuelle) pour créer les données.

5 Établir un lien entre les résultats observés et les quantités de mémoire vive et de mémoire virtuelle installées sur votre machine.

Pour n suffisamment petit pour que les données tiennent en mémoire vive, les temps obtenus seront inférieurs au millième de seconde (soit moins d'un millionième de seconde par accès). Enfin, pour n suffisamment petit pour que les données puissent tenir en mémoire virtuelle et suffisamment grand pour qu'assez de données soient envoyées dans le *swap*, on obtient des temps de l'ordre de la seconde (soit de l'ordre du millième de seconde par accès).

On donne figure A.2 des temps relevés par ce programme pour un PC sous une distribution GNU/Linux 32 bits avec 2 Go de mémoire physique plus 2 Go de mémoire virtuelle. Les valeurs de n ont été choisies aléatoirement de façon uniforme entre 400×10^6 et 700×10^6 mots-mémoire. D'autres programmes étaient ouverts sur ce PC, mais globalement, seul le programme de test était réellement actif. Résultat : les données et programmes non actifs ont été mis dans l'espace d'échange et les temps d'accès ne sont dégradés sensiblement que pour une quantité de données supérieure à 500 millions de mots, soit environ 2 Go (précisément la taille de la mémoire vive).

4. http://informatique-en-prepas.fr

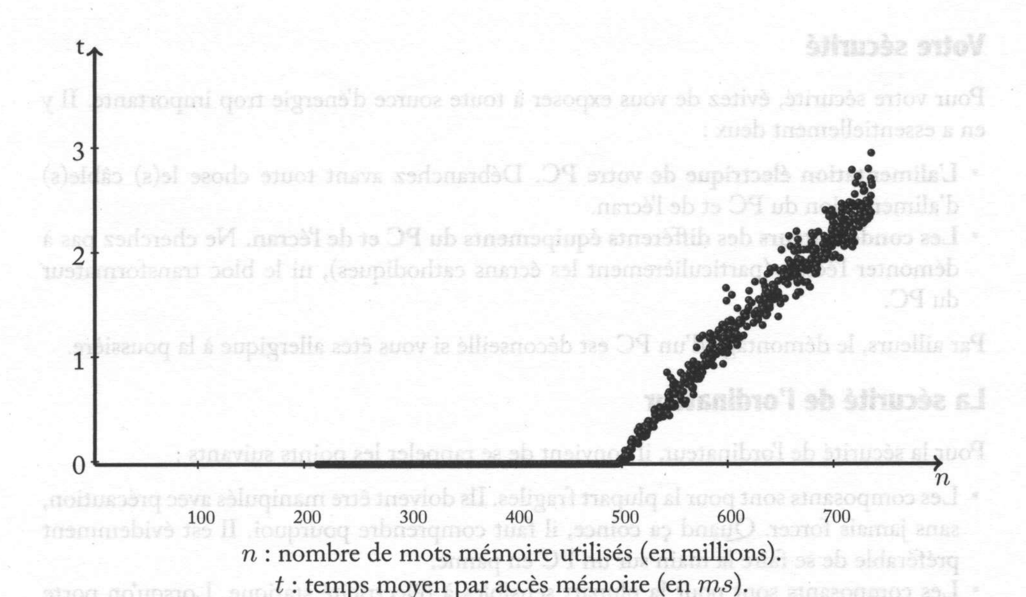


Figure A.2
Temps relevés par mem-virtuelle.py

A.3 Démontage d'un PC de bureau

Le but de ce TP est d'apprendre à repérer les différents composants matériels d'un PC de bureau et de voir comment changer un disque dur, une carte d'extension ou le processeur.

A.3.1 Sécurité

> **ATTENTION Avant de commencer**
>
> Il convient évidemment de prendre toutes les précautions nécessaires :
>
> - pour votre sécurité ;
> - pour éviter d'endommager l'ordinateur.
>
> Les informations que l'on donne ici sont des points à ne pas négliger. Avec un minimum de bon sens, le démontage d'un PC est sans danger pour celui qui l'effectue, mais manipuler un appareil électrique, quel qu'il soit, comporte des risques mortels. En cas de doute, le mieux est de chercher le conseil de personnes qualifiées.

Votre sécurité

Pour votre sécurité, évitez de vous exposer à toute source d'énergie trop importante. Il y en a essentiellement deux :

- **L'alimentation électrique de votre PC.** Débranchez avant toute chose le(s) câble(s) d'alimentation du PC et de l'écran.
- **Les condensateurs des différents équipements du PC et de l'écran.** Ne cherchez pas à démonter l'écran (particulièrement les écrans cathodiques), ni le bloc transformateur du PC.

Par ailleurs, le démontage d'un PC est déconseillé si vous êtes allergique à la poussière.

La sécurité de l'ordinateur

Pour la sécurité de l'ordinateur, il convient de se rappeler les points suivants :

- Les composants sont pour la plupart fragiles. Ils doivent être manipulés avec précaution, sans jamais forcer. Quand ça coince, il faut comprendre pourquoi. Il est évidemment préférable de se faire la main sur un PC en panne.
- Les composants sont pour la plupart sensibles à l'électricité statique. Lorsqu'on porte certains vêtements propices à l'accumulation d'électricité statique, on peut ressentir une étincelle électrique lorsqu'on touche le sol, un radiateur ou même simplement une autre personne. Ce genre d'étincelle est sans danger pour les humains, mais est fatale à de nombreux composants électroniques. Il convient donc, dans la mesure du possible, d'une part de diminuer les risques d'accumulation (pas de vêtements propices, pas de moquette) et d'autre part, de se décharger de son électricité statique avant de manipuler le PC (toucher n'importe quelle partie *métallique, non recouverte de peinture* de la carcasse du PC avant *chaque* manipulation). Les composants enlevés de la machine seront déposés sur une table propre.

A.3.2 Repérage des composants

Le support de la plupart des composants électroniques du PC est appelé la *carte mère*. Voir figure A.3 et figure A.4 une carte mère vue de face et de dos. C'est cette carte imprimée qui joue le rôle du bus de l'architecture de von Neumann (assistée cependant du *chipset*, ensemble de composants qui régule les transferts de données entre les composants du PC).

Figure A.3
Carte mère, vue de dessus

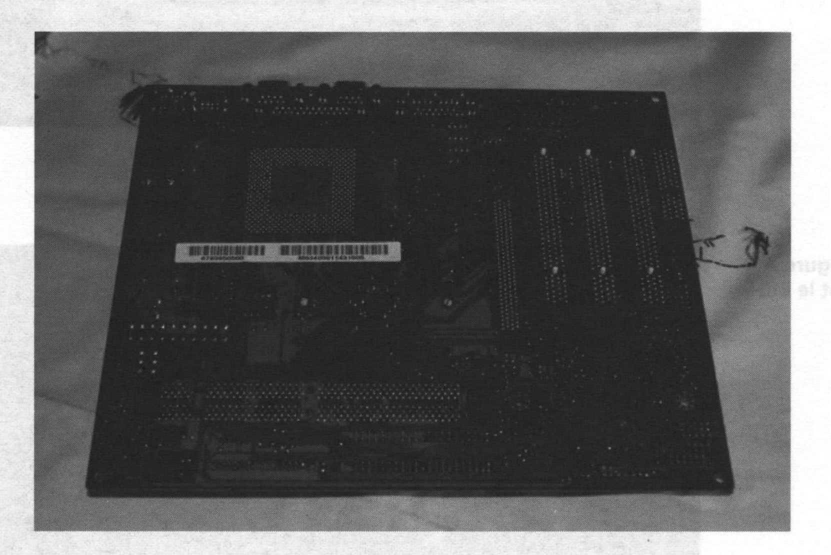


Figure A.4
Carte mère, vue de dessous

Sur cette carte mère, on distingue notamment le processeur (figure A.5) et la mémoire (ROM) sur laquelle se trouve le BIOS, programme régissant le comportement basique de l'ordinateur (figure A.6). On voit à l'avant-plan trois fiches pour connecter des nappes de disques dur (de trois couleurs différentes) et, immédiatement derrière, deux fiches pour les cartes de mémoire vive (figure A.7). Un peu plus à gauche se trouve l'emplacement de la pile bouton (utilisée pour alimenter l'horloge du PC lorsqu'il est éteint, voir figure A.8). À gauche, trois connecteurs sont disponibles pour brancher des cartes d'extensions (figure A.9).

Figure A.5
Processeur, en place sur la carte mère

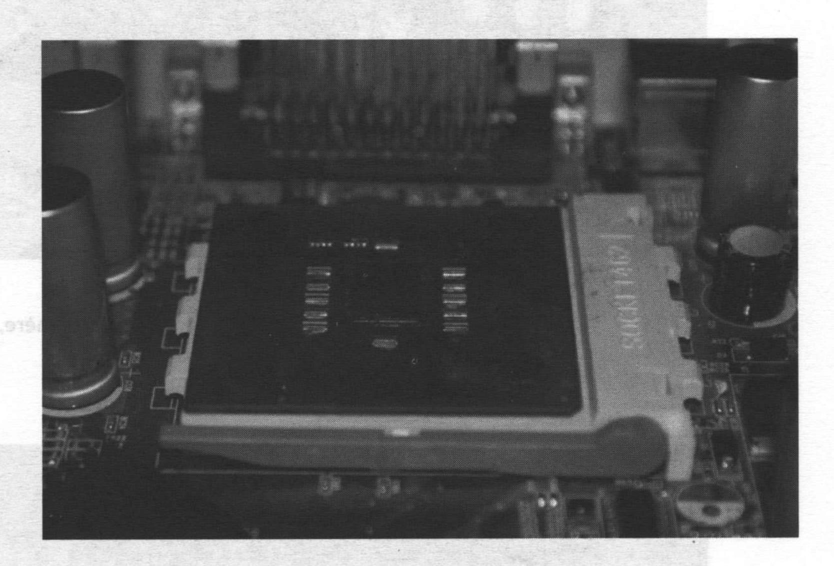


Figure A.6
Puce contenant le BIOS



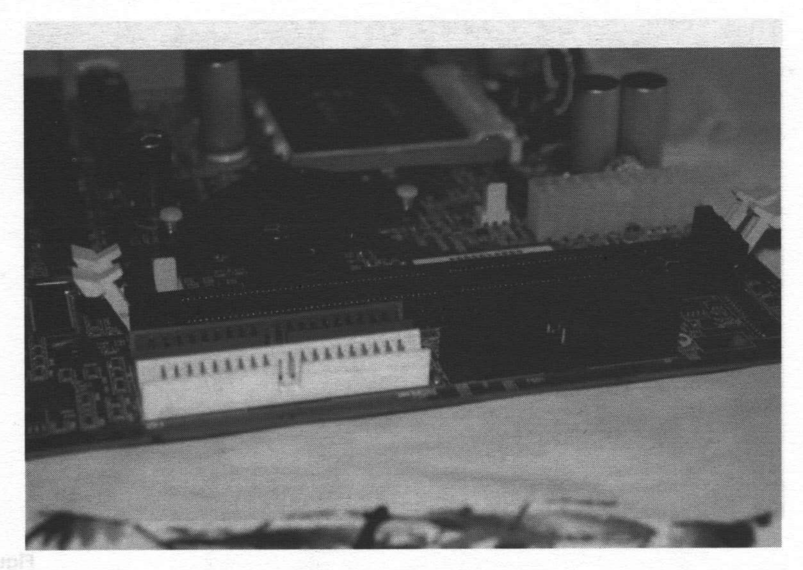


Figure A.7
Fiches de connexion pour les nappes de disques dur (colorées), fiches pour les barrettes de RAM (munies de loquets latéraux blancs)

Figure A.8
Emplacement de la pile bouton

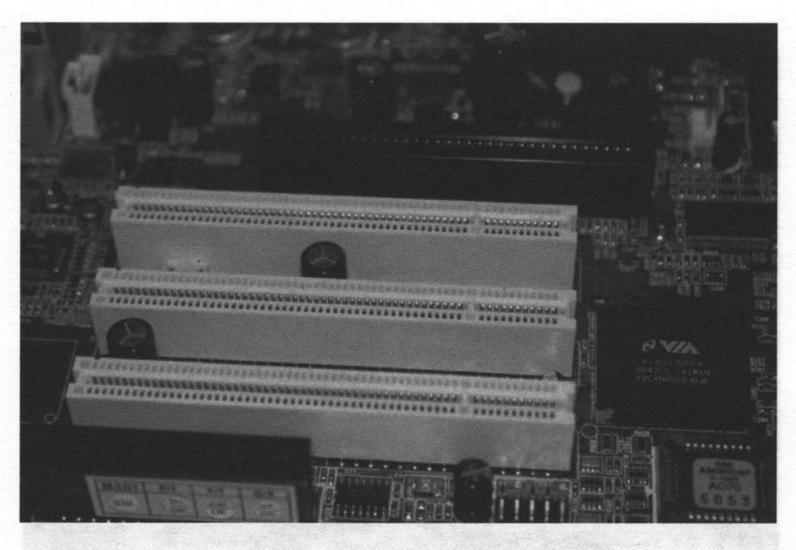


Figure A.9
Connecteurs pour les cartes d'extension

Retrait du processeur

Le processeur peut être enlevé. Pour cela, il convient de manœuvrer le loquet qui le bloque sur une fiche appelée *socket* et qui sert à réaliser les contacts électriques entre le processeur et la carte mère. Une fois le loquet relevé, on peut enlever le processeur (voir figure A.10). On constate que celui-ci est muni d'un grand nombre de petites pattes (figure A.11), qui permettent la communication avec la carte mère.

On peut également remarquer que le motif des trous sur le socket n'est pas tout à fait symétrique. C'est délibéré : cela assure qu'il n'y a qu'une seule façon d'y enficher le processeur : la bonne (on dit qu'il y a un *détrompeur*).

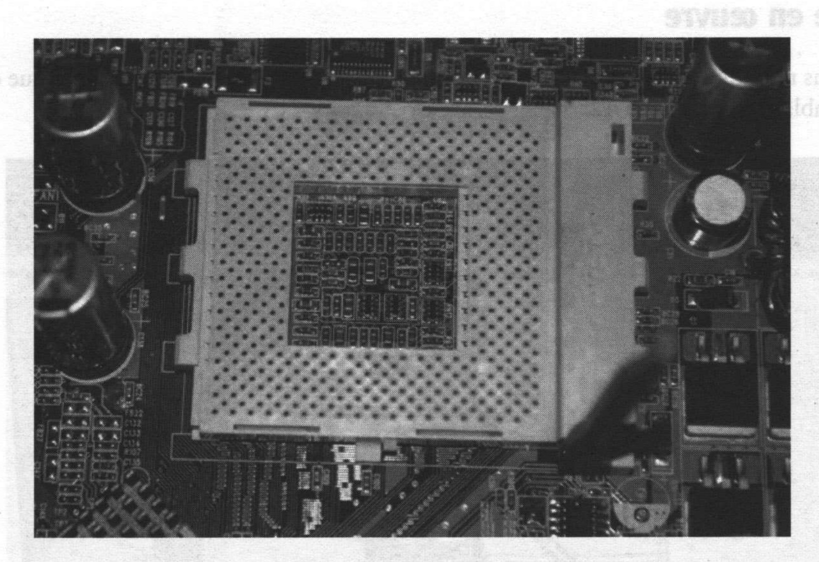


Figure A.10
Socket, loquet relevé

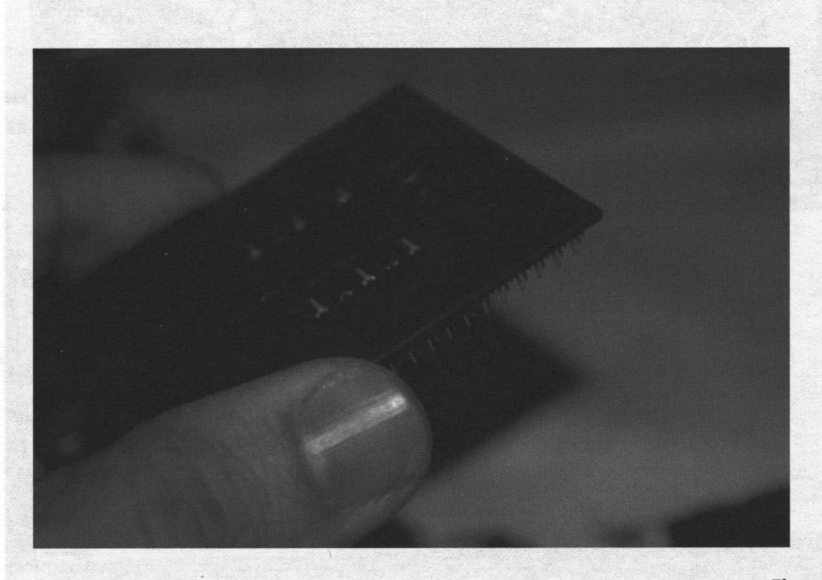


Figure A.11
Processeur, sorti de son socket

A.3.3 Mise en œuvre

Passons maintenant à un vrai PC. Une fois le côté du PC ouvert, on trouve quelque chose ressemblant à la figure A.12.

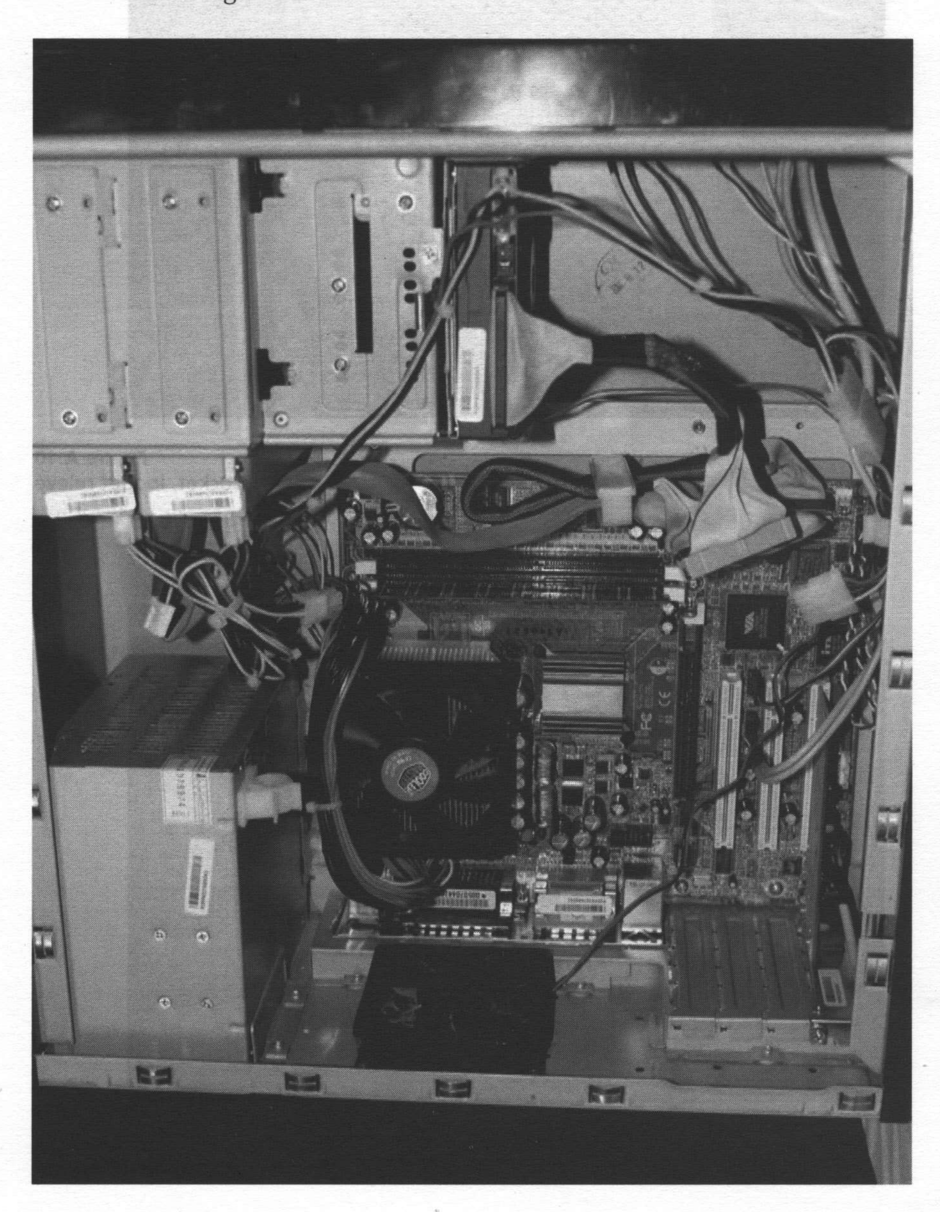


Figure A.12
Vue globale d'une unité centrale après ouverture du panneau latéral. L'arrière de l'unité se trouve à gauche.

Repérage

On peut repérer :

- **Le bloc d'alimentation.** Il est en haut à gauche et il en sort un faisceau de fils électriques. Une partie de ces câbles est reliée à la carte mère (ici à gauche du gros ventilateur).
- **Des ventilateurs.** Un est situé sur la carcasse à l'arrière de l'unité. Quelques fils d'alimentation de commande viennent de la carte mère jusqu'à lui. Un autre, énorme, est sur la carte mère, au-dessus d'un bloc métallique ressemblant à un radiateur. Il s'agit effectivement d'un radiateur, destiné à dissiper la chaleur du processeur, aidé par le ventilateur qui assure une circulation d'air.
- **Trois connecteurs d'extension.** Un seul, celui situé le plus bas, est occupé par un circuit imprimé.
- **La RAM.** Elle se présente sous forme de barrettes enfichables. Ici, on voit deux emplacements pour des barrettes mémoire (verticaux). Seul celui de gauche est occupé, l'autre est libre, ce qui laisse penser qu'on pourrait augmenter la capacité de ce PC en mémoire vive par l'ajout d'une autre barrette.
- **La pile bouton.** Où est-elle ?
- **Des sortes de ruban.** Ils partent de la carte mère en bas à droite et la relient au lecteur de DVD et au graveur de CD situés en haut à droite, ainsi qu'au disque dur (exactement à mi-hauteur, tout à droite).

Changement du disque dur

Le disque dur a la forme d'un boîtier, généralement de dimensions 101 mm × 146 mm × 25, 4 mm. Sa face supérieure comporte un circuit imprimé sur lequel se trouvent plusieurs puces (voir figure A.13). Il est connecté d'une part au bloc d'alimentation par des fils électriques (en général rouges, jaunes et noirs) et à la carte mère par une nappe (voir figure A.14). Il est fixé au boîtier dans un berceau métallique par des vis sur son côté.

Changer le disque dur est extrêmement simple : il suffit de débrancher l'alimentation ainsi que la nappe, de dévisser les vis qui le retiennent, d'enlever le disque dur et de le remplacer par un autre.

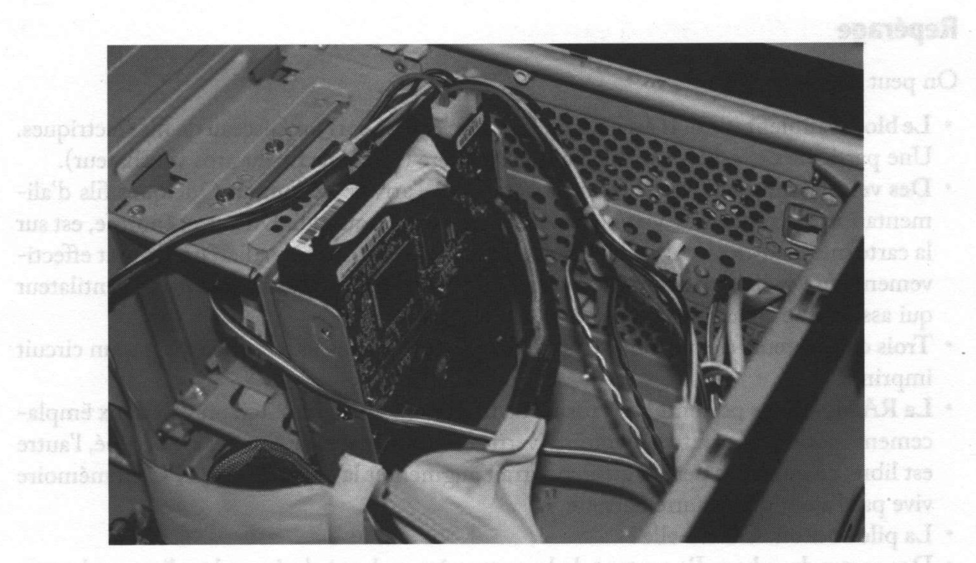


Figure A.13
Le disque dur dans son berceau

Figure A.14
Le branchement des nappes sur la carte mère (la nappe visible est celle du lecteur de DVD et cache presque totalement celle du disque dur)

Changement d'une carte d'extension

Les cartes d'extension servent à apporter de nouvelles fonctionnalités matérielles au PC. Le plus souvent, il s'agit de cartes réseau (cartes ethernet), qui sont maintenant intégrées sur la carte mère. Il arrive cependant qu'on veuille ajouter une deuxième carte réseau ou simplement qu'on veuille pouvoir de nouveau avoir une connexion ethernet lorsque la carte intégrée est en panne.

Ici, la carte du PC (voir figure A.15) est une carte MODEM, permettant de se connecter au réseau téléphonique pour émettre des appels, envoyer/recevoir des fax ou se connecter à Internet (ou un autre réseau) par le réseau téléphonique. Pour la retirer, il suffit de dévisser la vis qui la maintient au boîtier (voir figure A.16) puis de la sortir de son logement.

> **EN PRATIQUE Manipuler une carte d'extension**
> Il ne faut jamais toucher les composants sur la carte d'extension, et toujours préférer la tenir par les côtés.

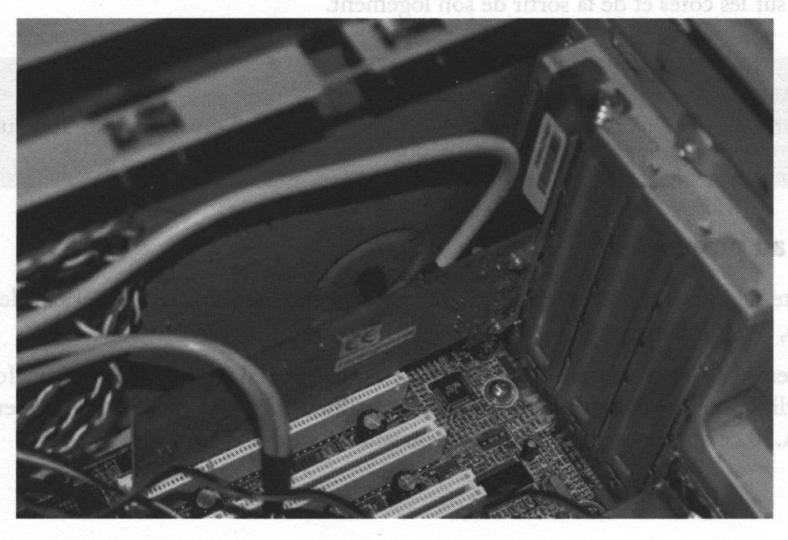


Figure A.15
Carte MODEM en place dans l'ordinateur.

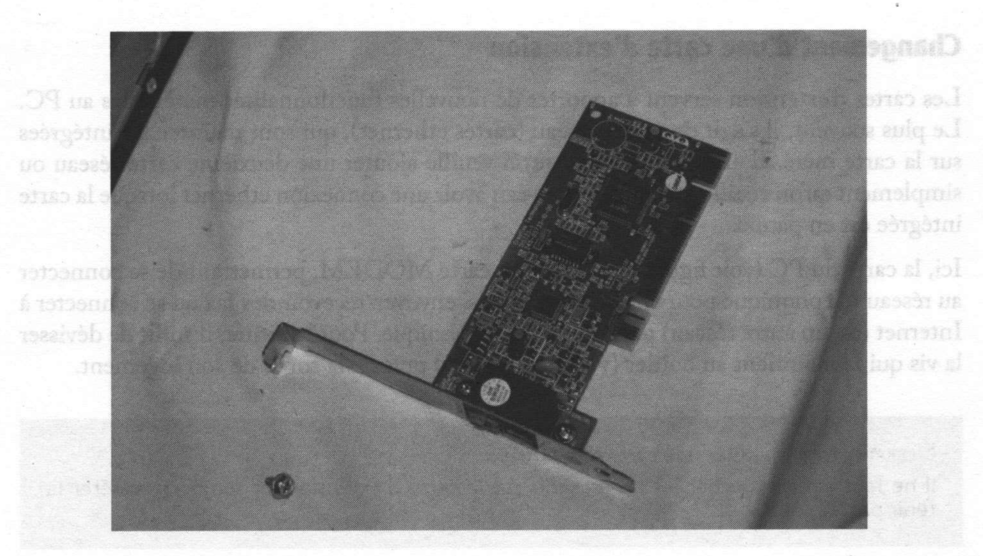


Figure A.16
Carte MODEM, retirée.

Changement des barrettes mémoire

La barrette mémoire (figure A.17) peut aisément être retirée. Il suffit d'ouvrir les deux loquets sur les côtés et de la sortir de son logement.

> **EN PRATIQUE Manipuler une barette mémoire**
> La barrette est un composant particulièrement fragile et sensible à l'électricité statique. Ne la prendre que sur les côtés du circuit intégré.

Accès au processeur

On tente maintenant d'aller retirer le processeur. Pour tout travail à l'intérieur de l'unité centrale, il est préférable de la coucher.

Il convient d'abord d'enlever le ventilateur. Il est en général fixé sur le radiateur. Ici, il est en fait clipsé sur une pièce plastique qui semble difficilement séparable du processeur. Voir figure A.18 ce que ça donne après retrait du ventilateur.

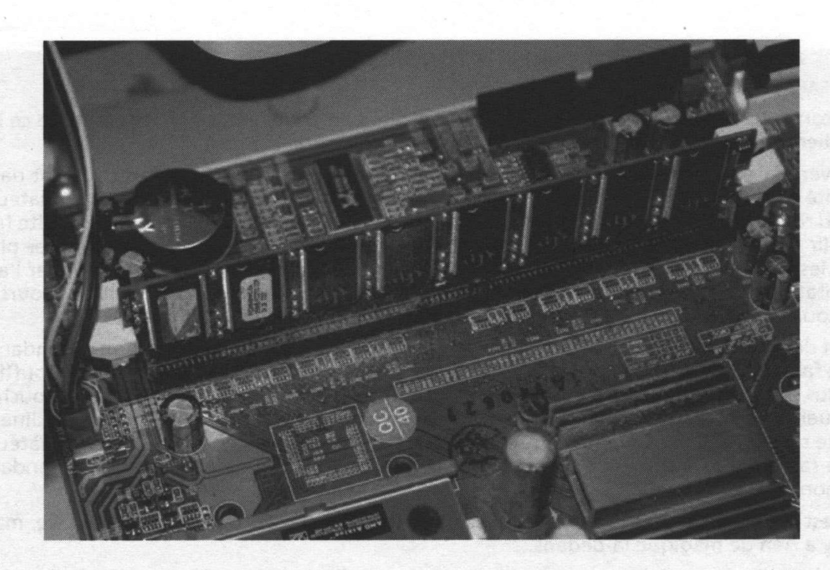


Figure A.17
Barrette de RAM (on voit distinctement les 8 puces constituant autant de modules mémoire)

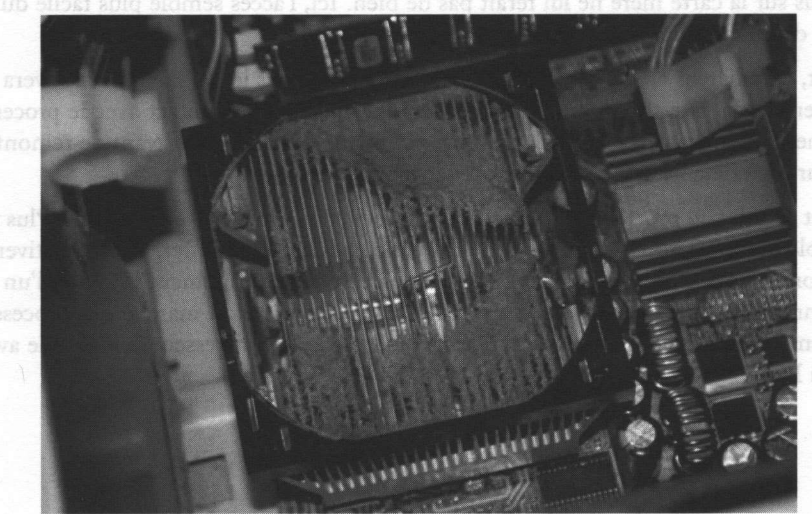


Figure A.18
Le radiateur du processeur

En pratique Que de poussières !

On constate que le ventilateur figure A.18 est plein de poussière. Le propriétaire de ce PC oublierait-il de faire le ménage chez lui ? Non, l'explication est tout autre.

Les ventilateurs brassant de l'air en permanence, les poussières ambiantes circulent dans l'unité centrale et s'y accumulent au fil du temps, en particulier au niveau des ventilateurs et du radiateur, réduisant leur efficacité. L'augmentation de température qui en résulte fait vieillir plus vite les composants, conduit les capteurs de température à faire tourner plus vite les ventilateurs (le niveau de bruit peut augmenter sensiblement) voire à couper l'alimentation en cas de surchauffe trop importante. Enfin, les poussières peuvent être sources de courts-circuits qui empêchent complètement la machine de fonctionner.

C'est donc une bonne idée de nettoyer l'intérieur d'un PC de bureau. Attention cependant : il ne faut pas toucher à la carte mère, car cela risque de l'endommager. Il suffira de souffler dessus, soit avec un appareil adapté (un sèche-cheveux est inefficace), soit à la bouche, auquel cas mieux vaut fermer les yeux et ne pas craindre la poussière (les auteurs déclinent toute responsabilité). On peut en revanche gratter la poussière déposée sur les ventilateurs et le radiateur sans danger. Pour atteindre le radiateur du processeur, il faudra cependant démonter le ventilateur.

Il n'est pas rare que souffler sur l'intérieur d'une unité centrale ramène le PC à la vie, mais il n'y a rien de magique là-dedans...

Dernière chose : le problème est le même pour un ordinateur portable, mais le démontage et le remontage en sont beaucoup plus délicats que pour un PC de bureau...

Il faut maintenant enlever le radiateur. Il est en général fixé sur le socket par un clip métallique. Il est difficile d'y accéder. On y parvient en général à l'aide d'un tournevis. Il faut une certaine force pour appuyer sur le clip, mais éviter absolument de déraper : un coup de tournevis sur la carte mère ne lui ferait pas de bien. Ici, l'accès semble plus facile du côté du bloc d'alimentation (figure A.19)

Ça y est, le radiateur est retiré (figure A.20). Attention, sous le radiateur on trouvera probablement de la pâte thermique, produit assurant un meilleur contact avec le processeur et augmentant le transfert de chaleur. Il est préférable d'en remettre avant de remonter le radiateur (et à défaut, de ne pas enlever celle en place).

On peut accéder au processeur, le sortir de son socket et le remplacer si besoin. Plus vraisemblablement, on se contentera de nettoyer le radiateur. En effet, il est relativement rare qu'on remplace un processeur, d'une part parce que c'est rarement la cause d'un dysfonctionnement du PC et d'autre part, parce que l'évolution du marché des processeurs est tellement rapide qu'il est souvent difficile de trouver un processeur compatible avec le reste du PC.

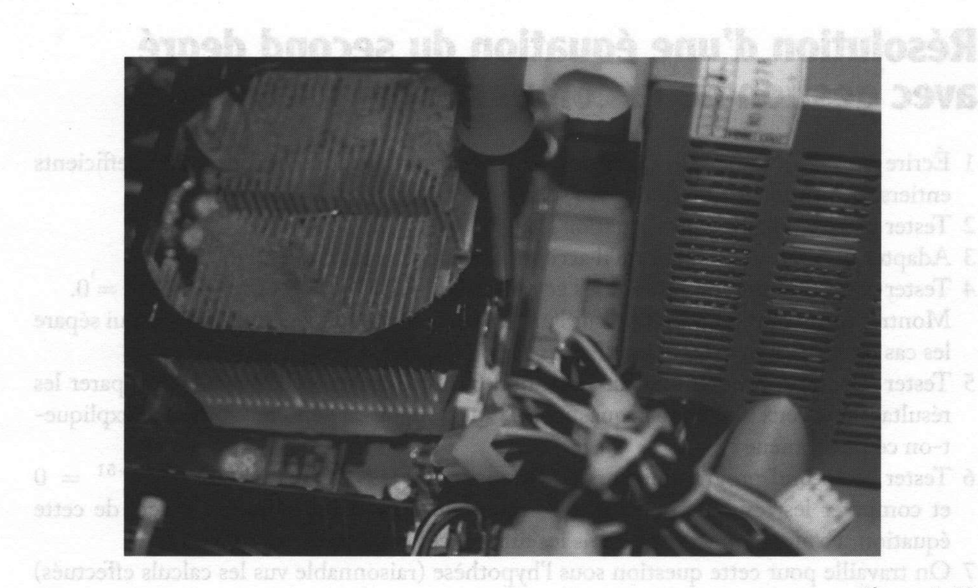


Figure A.19
Retrait du clip du radiateur. Les picots (en blanc) du socket étaient enfichés dans les trous de la fiche métallique du radiateur.

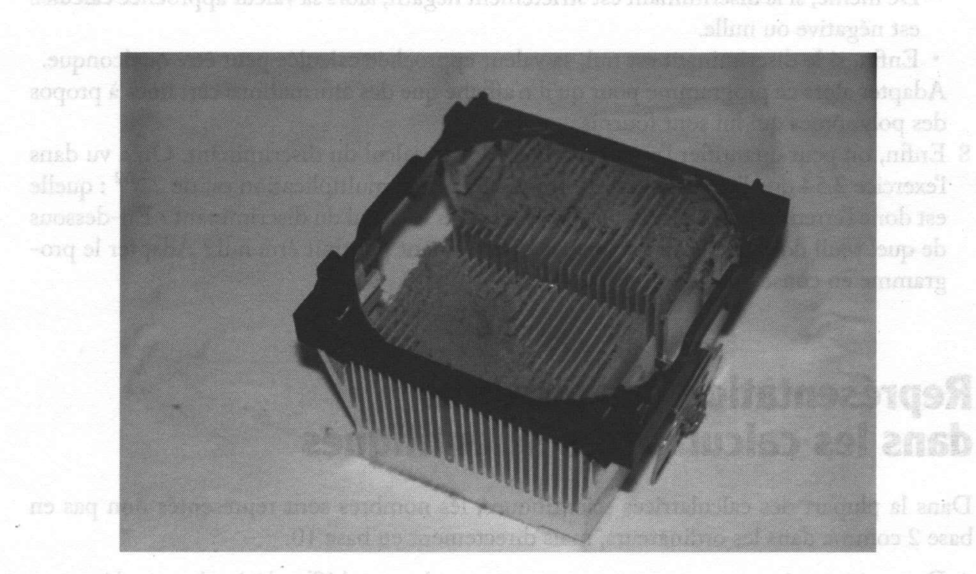


Figure A.20
Radiateur du PC, avant nettoyage.

A.4 Résolution d'une équation du second degré avec gestion de la comparaison à zéro

1 Écrire un programme qui, étant donnée une équation du second degré à coefficients entiers, détermine le nombre de ses solutions et leurs valeurs éventuelles.
2 Tester ce programme pour l'équation $x^2 + 6x + 9 = 0$.
3 Adapter ce programme pour qu'il accepte des coefficients non entiers.
4 Tester ce second programme sur les équations $x^2 + 10^{-10} = 0$ et $x^2 - 10^{-10} = 0$. Montrer qu'une infime variation sur l'un des coefficients fait franchir la ligne qui sépare les cas où l'équation a des solutions des cas où elle n'en a pas.
5 Tester le second programme sur l'équation $0,1x^2 + 0,6x + 0,9 = 0$. Comparer les résultats avec ceux obtenus à la question 2. Que doit-on en penser ? Comment explique-t-on ce phénomène ?
6 Tester ce second programme sur l'équation $x^2 + (1 + 2^{-50})x + 0,25 + 2^{-51} = 0$ et comparer les résultats obtenus avec ceux que prévoit la résolution exacte de cette équation. Expliquer encore une fois les différences constatées.
7 On travaille pour cette question sous l'hypothèse (raisonnable vus les calculs effectués) que la norme IEEE 754 assure les propriétés suivantes des arrondis :
   - Si le discriminant est strictement positif, alors sa valeur approchée calculée est positive ou nulle.
   - De même, si le discriminant est strictement négatif, alors sa valeur approchée calculée est négative ou nulle.
   - Enfin, si le discriminant est nul, sa valeur approchée calculée peut être quelconque.

   Adapter alors ce programme pour qu'il n'affiche que des affirmations certaines à propos des polynômes qui lui sont fournis.
8 Enfin, on peut quantifier l'erreur commise sur le calcul du discriminant. On a vu dans l'exercice 2.54 que l'erreur relative introduite par une multiplication est de 2^{-52} : quelle est donc l'erreur relative maximale commise dans le calcul du discriminant ? En-dessous de quel seuil doit-on considérer que le discriminant pourrait être nul ? Adapter le programme en conséquence.

A.5 Représentation des nombres dans les calculatrices scientifiques

Dans la plupart des calculatrices scientifiques, les nombres sont représentés non pas en base 2 comme dans les ordinateurs, mais directement en base 10.

1 Dans un premier temps, on peut supposer que chaque chiffre décimal est codé sur un groupe de bits distinct. Combien de bits sont nécessaires pour coder un chiffre décimal ? Combien de nombres différents peut-on coder en binaire sur ce même nombre de bits ?

2 Afin d'obtenir une représentation en mémoire plus efficace, les constructeurs de calculatrices regroupent parfois trois chiffres décimaux sur un même groupe de bits. Expliquer pourquoi ce choix est a priori meilleur que celui de la question précédente. Imaginer également les problèmes que cela peut poser.

3 Les dépassements arithmétiques sur les entiers existent-ils sur votre calculatrice ? Pour le vérifier, on pourra par exemple essayer de calculer et d'afficher les puissances de 2 successives : au bout d'un certain nombre d'itérations, la façon dont le nombre calculé s'affiche est différente. Que peut-on en conclure quant à la représentation des nombres dans la mémoire des calculatrices [5] ?

4 On va maintenant étudier la représentation en machine des nombres à virgule flottante.

a) Dans un premier temps, on peut déterminer la taille de la mantisse. Pour cela, programmer ou exécuter à la main l'algorithme suivant, en prenant bien soin d'écrire `1.0` à la première instruction pour assurer que a sera un nombre à virgule flottante :

$a \leftarrow 1.0$
tant que $a + 1.0 - a = 1.0$ **faire**
  $a \leftarrow 2.0 * a$
Résultat : a

Expliquer pourquoi la condition de la boucle **tant que** finit par être fausse et déduire de la valeur finale de a la taille de la mantisse en mémoire.

b) Ensuite, on peut évaluer le nombre de chiffres sur lequel l'exposant est représenté. Pour cela, reprendre l'algorithme précédent avec une boucle **tant que** dont la condition est $2.0 * a \neq a$. Pour quelle raison cet algorithme finit-il également par s'arrêter ? Déduire de la dernière valeur prise par a la taille de l'exposant en mémoire. Vérifier qu'une limite similaire existe pour les exposants négatifs.

c) Les calculatrices possèdent pour la plupart des fonctions qui permettent de vérifier l'état de la mémoire. À l'aide de ces fonctions, déterminer la place utilisée par la variable a en mémoire : si cette valeur est cohérente avec les résultats des deux questions précédentes, quelle est l'unité de taille mémoire utilisée ici ?
Sur les calculatrices permettant d'écrire des entiers longs, on vérifiera également la croissance de la taille mémoire utilisée par un entier en fonction de sa valeur.

5. Certaines calculatrices permettent de faire des calculs symboliques ; elles disposent alors d'entiers longs similaires à ceux de Python. Une lecture approfondie de la documentation révèle tout de même que le nombre de chiffres autorisé est borné, par exemple 614 chiffres sur les TI 89 et 92. Un algorithme similaire à celui de la prochaine question avec des valeurs entières met en lumière cette limitation.

A.6 Arithmétique et cryptographie

A.6.1 Algorithme d'Euclide

Algorithme d'Euclide originel

Soient $u, v \in \mathbb{N}$. L'algorithme suivant, dit *algorithme d'Euclide*, calcule le plus grand diviseur commun (PGCD) de u et v :

A1 Si $v = 0$ alors la réponse est u.
A2 Faire $(u, v) \leftarrow (v, u \bmod v)$. Retourner en **A1**.

Écrire une fonction `euclide` implantant cet algorithme.

Complexité

La complexité de l'algorithme d'Euclide est donnée par le résultat suivant :
Théorème (G. Lamé, 1845). Si $0 \leqslant u, v < N$, le nombre de divisions dans l'algorithme d'Euclide appliqué à u et v est au plus $\lceil \log_\phi(\sqrt{5}N) \rceil - 2$, où ϕ est le nombre d'or $\frac{1+\sqrt{5}}{2}$.

Algorithme d'Euclide étendu

Soient $u, v \in \mathbb{N}$. L'algorithme d'Euclide peut être adapté pour calculer, en même temps que le PGCD de u et v, les coefficients de Bezout. L'algorithme suivant calcule un triplet (u_1, u_2, u_3) tel que $uu_1 + vu_2 = u_3 = u \wedge v$.

B1 $(u_1, u_2, u_3) \leftarrow (1, 0, u)$.
B2 $(v_1, v_2, v_3) \leftarrow (0, 1, v)$.
B3 Si $v_3 = 0$ alors la réponse est (u_1, u_2, u_3).
B4 Soit $q = \lfloor u_3 / v_3 \rfloor$. Faire
$(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - q(v_1, v_2, v_3)$
$(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$
$(v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$
Retourner en B3.

Écrire une fonction `bezout` qui réalise cet algorithme.

Démontrer la correction de cette fonction à l'aide d'un invariant.

Application : division modulo m

Soient $u, v, m \in \mathbb{N}^\star$ tels que $v \wedge m = 1$. On appelle quotient de u par v modulo m tout entier w tel que $0 \leqslant w < m$ et $u \equiv vw \pmod{m}$.

Écrire une fonction calculant le quotient de u par v modulo m.

A.6.2 Décomposition en facteurs premiers

Étant donné $n \in \mathbb{N}$, n s'écrit de manière unique sous la forme :

$$n = p_1 p_2 \dots p_k \qquad \text{avec} \quad p_1 \leqslant p_2 \leqslant \dots \leqslant p_k \quad \text{et} \quad p_i \text{ premier}$$

On se propose ici de déterminer les facteurs p_i.

Méthode par division

La méthode la plus simple est la suivante : si $n > 1$, on teste sa divisibilité par les nombres premiers successifs $p = 2, 3, 5, \dots$ jusqu'à ce que $n \equiv 0 \pmod p$. On remplace alors n par n/p et on reprend à p. Lorsque $n \not\equiv 0 \pmod p$ avec $\lfloor n/p \rfloor \leqslant p$, on s'arrête, avec n premier.

Cette méthode a l'inconvénient qu'il faut déterminer la suite des nombres premiers. Toutefois, on peut simplifier cette méthode de la façon suivante. Soit $n \in \mathbb{N}^*$ dont on souhaite déterminer la décomposition en facteurs premiers. Soit (d_i) une suite d'entiers

$$2 = d_0 < d_1 < d_2 < \dots$$

qui inclut tous les nombres premiers $\leqslant \sqrt{n}$ et au moins un élément $d_k \geqslant \sqrt{n}$. Alors, l'algorithme suivant détermine la décomposition de n en facteurs premiers :

A1 $t \leftarrow 0$, $k \leftarrow 0$.
A2 Si $n = 1$ c'est terminé.
A3 On divise n par d_k : $n = qd_k + r$ avec $(0 \leqslant r < d_k)$.
A4 Si $r = 0$ alors $t \leftarrow t + 1$, $p_t \leftarrow d_k$, $n \leftarrow q$. Aller en A2.
A5 Si $q > d_k$ alors $k \leftarrow k + 1$ et aller en **A3**.
A6 $t \leftarrow t + 1$, $p_t \leftarrow n$. C'est terminé.

Écrire une fonction **decomp** prenant en argument l'entier n, une suite (d_k) pour n, et renvoyant la suite croissante des facteurs premiers p_i de n.

On pourrait prendre pour (d_k) la suite $2, 3, 5, 7, \dots$, c'est-à-dire 2 puis tous les impairs à partir de 3. On prendra plutôt la suite $2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \dots$, c'est-à-dire ajouter alternativement 2 et 4 à partir de 5 (on supprime ainsi tous les multiples de 2 et 3).

Écrire une fonction **dk** prenant n en argument et renvoyant une telle suite $d_0, \dots, d_k$ avec $d_k \geqslant \sqrt{n}$.

On peut gagner encore 20 % sur cette suite en supprimant les entiers de la forme $30m \pm 5$, et encore 14 % en supprimant les multiples de 7, etc. Si n est petit, on peut utiliser une table des nombres premiers (pour $n \leqslant 10^6$, il n'y a que 168 nombres premiers).

Complexité

Déterminer la complexité de cet algorithme est un problème très difficile, mais en pratique l'algorithme n'est plus guère utilisable au-delà de 10^6.

Méthode de Fermat

On se propose ici de réaliser un autre algorithme de décomposition en facteurs premiers, dû à Pierre de Fermat (1643), plus adapté à la recherche de grands facteurs premiers.

Soit N impair s'écrivant sous la forme uv, avec $u \leqslant v$. On définit alors :

$$x = (u+v)/2, \qquad y = (v-u)/2$$

et on a :

$$N = x^2 - y^2, \qquad 0 \leqslant y < x \leqslant N$$

La méthode de Fermat consiste à rechercher des valeurs de x et y satisfaisant les conditions ci-dessus. Étant donné N impair, l'algorithme suivant détermine le plus grand facteur de N inférieur ou égal à $\sqrt{N}$:

A1 Initialiser : $x' \leftarrow 2\lfloor\sqrt{N}\rfloor + 1, \quad y' \leftarrow 1 \quad$ et $\quad r \leftarrow \lfloor\sqrt{N}\rfloor^2 - N$
(dans la suite, x' correspond à $2x+1$, y' à $2y+1$ et r à x^2+y^2-N).
A2 Si $r > 0$ alors faire $r \leftarrow r - y'$, $y' \leftarrow y' + 2$. Aller en **A2**.
A3 Si $r < 0$ alors faire $r \leftarrow r + x'$, $x' \leftarrow x' + 2$. Aller en **A2**.
A4 Si $r = 0$ alors c'est terminé : on a

$$N = ((x'-y')/2)((x'+y'-2)/2)$$

et $(x'-y')/2$ est le plus grand facteur de N inférieur ou égal à $\sqrt{N}$.

Écrire une fonction `fermat` réalisant l'algorithme ci-dessus et renvoyant la décomposition en deux facteurs obtenue. En déduire une nouvelle fonction `decomp2` de décomposition en facteurs premiers.

A.6.3 Recherche de grands nombres premiers

Pour déterminer si un nombre est premier, on peut évidemment utiliser les fonctions `decomp` ou `fermat` définies précédemment. Donner une estimation (grossière) de l'entier maximum dont on peut vérifier qu'il est premier avec ces méthodes en un temps de moins de dix secondes. Pour vérifier que cette estimation est correcte, on pourra essayer les fonctions `decomp` et `fermat` sur les nombres $2^p - 1$, en prenant pour p les valeurs successives 17, 19, 31, 61, 107, 127.

On s'attache maintenant à trouver une méthode plus efficace pour tester la primalité d'un entier. On s'intéressera ici au test de Fermat.

Test de Fermat

Soit n un entier. On dit que n passe le test de Fermat de base a si $a^{n-1} \equiv 1[n]$. On sait que si n est premier, alors pour tout entier $a \in]0, n[$, n passe le test de Fermat de base a. Donc, par contraposée, si n ne passe pas le test de Fermat de base a, alors n n'est pas premier. On dit dans ce cas que a est un témoin de non-primalité de n pour le test de Fermat. Si n passe le test de Fermat de base a mais n'est pas un nombre premier, on dit que n est un pseudo-premier de Fermat de base a.

Écrire une fonction `passe_fermat(a,n)` à valeur booléenne indiquant si n passe le test de Fermat de base a.

Y a-t-il des entiers compris entre 1 et 1 000 qui soient des pseudo-premiers de Fermat de base 2 (on les appelle aussi nombres de Poulet) ? Lesquels ?

Y a-t-il des entiers n compris entre 1 et 1 000 qui soient des pseudo-premiers de Fermat de base a pour tout a ? Pour tout a appartenant à $]0, n[$?

Pour tester si a^{n-1} est congru à 1 modulo n, on peut calculer `a**(n-1) % n`. Combien de temps ce calcul prend-il pour $n = 100\,000\,000$ et $a = 2$? Python propose une fonction spécialisée pour effectuer ce calcul : `pow`. On peut calculer `pow(a, n-1, n)`. Comparer avec le temps d'exécution précédent.

Les nombres premiers sont très utiles en cryptographie. Le logiciel de chiffrement PGP utilise ainsi les tests de Fermat de bases 2, 3, 5 et 7 pour décider si un nombre est premier [6].

Écrire une fonction `premier_PGP(n)` « testant » si un nombre est premier avec la méthode utilisée par PGP.

En considérant que ce test est parfait, écrire une fonction `premier_suivant(n)` rendant le plus petit nombre premier strictement supérieur à n.

Chercher expérimentalement jusqu'à quelle valeur de n on obtient une réponse en moins de dix secondes.

Test de Miller-Rabin

On propose maintenant un autre test, appelé test de Miller-Rabin. Il fonctionne de la façon suivante. Pour tester si un entier n est premier, on commence par écrire $n-1$ sous la forme $2^s \times m$, où m est impair. Soit $a \in [\![1, n-1]\!]$. Le test repose sur le résultat suivant : si n est premier, alors ou bien $a^m \equiv 1[n]$, ou bien il existe $d \in [\![0, s-1]\!]$ vérifiant $a^{2^d.m} \equiv -1[n]$. On peut démontrer ce résultat en utilisant le petit théorème de Fermat, le fait que si n est premier $\mathbb{Z}/n\mathbb{Z}$ est un corps et le fait que dans tout anneau intègre, l'équation $x^2 = 1$ a au plus deux solutions [7] (1 et -1).

6. Le risque de choisir accidentellement un nombre non premier est apparemment très faible pour les plages de nombres testées par PGP.

7. Dans certains anneaux intègres, il n'y en a qu'une, par exemple dans $\mathbb{Z}/2\mathbb{Z}$, où $-1 = 1$.

On dira que a est un *menteur fort* s'il vérifie les conditions précédentes et si n n'est pas premier.
On peut montrer qu'au plus un quart des valeurs de $[\![1, n-1]\!]$ sont des menteurs forts[8]. Le test de Miller-Rabin s'effectue donc en pratique de la façon suivante : prendre une valeur a au hasard comprise entre 1 et $n-1$ et effectuer le test avec cette base a. Si a respecte les conditions données ci-avant, n est probablement premier ; dans ce cas, on recommence avec un nouveau a pris au hasard. Si au bout de 40 essais, on n'a pas trouvé de a violant les conditions données plus haut, on considérera que le nombre donné était premier.
Si le nombre testé est premier, quelle est la probabilité que le test de Miller-Rabin le déclare non premier ? S'il n'est pas premier, majorer la probabilité que le test de Miller-Rabin le déclare premier.

Pour effectuer le test aussi vite que possible, on calculera le reste de a^m modulo n et on effectuera ensuite des mises au carré modulo n successives pour calculer a^{2m}, a^{4m},...,$a^{2^{s-1}m}$. De plus, on n'est pas obligé de calculer toutes ces valeurs : on s'arrête dès le début si $a^m \equiv 1[n]$, on s'arrête également dès qu'on trouve la valeur -1 et enfin, on peut aussi s'arrêter si l'on trouve la valeur 1. Pourquoi ?

Mettre en œuvre ce test pour vérifier si les grands nombres trouvés avec le test PGP sont bien premiers.

A.6.4 Application à la cryptographie : la méthode RSA

Parmi les procédés cryptographiques, la méthode RSA, découverte par R. Rivest, A. Shamir et L. Adleman en 1978, est l'une des plus utilisées actuellement. Elle fait partie des méthodes dites *à clé publique* : chaque personne possède une clé P *publique* dont tout le monde peut avoir connaissance (par exemple dans un annuaire) et une clé S *secrète* qu'elle seule connaît. Lorsqu'on veut envoyer un message M à une personne A, on le code avec la clé publique P_A du destinataire. Ce dernier le décode alors avec sa clé secrète S_A et peut le lire.

Un tel système fonctionne donc si on a les propriétés suivantes :

1 $S(P(M)) = M$ pour tout message M.
2 Les paires (S, P) sont toutes distinctes.
3 Découvrir la clé secrète S à partir de la clé publique P est aussi difficile que déchiffrer le message codé.
4 Une paire (S, P) peut se calculer facilement.

La méthode RSA fonctionne de la manière suivante : soient x, y et s trois grands nombres premiers, tels que $x, y \leqslant s$. Soient $N = xy$ et p tel que ps mod $(x-1)(y-1) = 1$. On peut alors montrer que pour tout M, on a $M^{ps} = M$ (mod N).

8. Voir sur Wikipédia en anglais l'article *Miller–Rabin primality test* et notamment l'article de René Schoof mentionné dans les références (article consulté le 22 mars 2013).

La clé publique est alors le couple (N, p) et la clé secrète le couple (N, s). Pour coder un message, on commence par le découper en entiers inférieurs à N et on élève alors ces entiers à la puissance p modulo N. Pour le décoder, on élève les entiers composant le message à la puissance s modulo N.

En supposant que l'on dispose d'un générateur de grands nombres premiers, comment engendrer un couple clé publique-clé secrète pour la méthode RSA ?

Un message est codé de la manière suivante. Chaque caractère est d'abord remplacé par son code ASCII (65 pour `A`, 66 pour `B`, etc.). Le message est ensuite chiffré en utilisant la clé publique (N, p) où $N = 49808911$ et $p = 5685669$. Le résultat est le suivant :

```
49583279, 4553592, 17767401, 16172223, 33062955, 33599637,
17767401, 11607763, 17767401, 9275561, 11607763, 35959722, 17767401,
44022065, 3148857, 17767401, 40136246, 13922222, 16172223, 17767401,
4553592, 11607763, 16172223, 25708244, 11607763, 25708244, 3148857,
36260425, 17767401, 35959722, 3148857, 29735027, 4553592, 3148857,
29538442, 3148857, 16172223, 31299930, 17767401, 44022065, 3148857,
17767401, 40136246, 13922222, 16172223, 35959722, 17767401, 9275561,
45114532, 13922222, 25708244, 45114532, 11607763, 29538442, 29538442,
3148857, 29735027, 45114532, 35959722
```

« Craquer » ce codage RSA pour découvrir le message en clair. On rappelle que la fonction `chr` renvoie le caractère dont on donne le code ASCII et que l'on peut afficher le caractère `c` sans qu'il soit suivi d'un retour chariot avec `print(c, end="")`.

Donner une idée des tailles de nombres premiers qu'il faut choisir pour ne plus pouvoir casser ce codage par factorisation de N. Il est à noter qu'il existe d'autres méthodes pour casser le codage choisi ici sans factoriser N. En pratique la mise en œuvre de RSA est donc plus complexe.

A.7 Manipulation d'images bitmap

Pour représenter une image telle que le cercle figure A.21, une possibilité est de lui superposer une grille, dont on appelle chaque case un *pixel* (*picture element*). On colorie ensuite les cases de la grille par lesquelles passe un arc du cercle. On obtient alors la figure de droite, qui est certes moins régulière que le cercle initial, mais en constitue déjà une première approximation correcte.

Si le résultat obtenu n'est pas satisfaisant, on peut recommencer le processus avec une grille plus fine, par exemple de 100 cases de côté, et l'approximation obtenue sera meilleure, voire indiscernable de l'original à partir d'une certaine résolution.

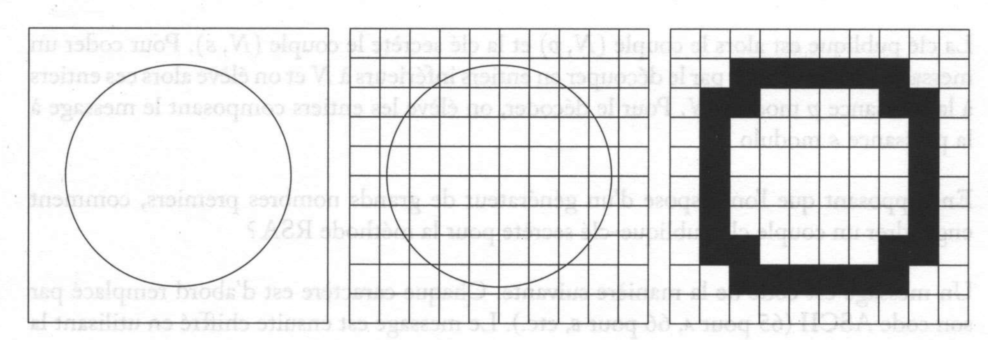


Figure A.21
Représentation d'une image par une matrice de bits

Cette méthode n'est bien sûr pas spécifique au cercle et fonctionne avec un tracé quelconque. On est donc en mesure de remplacer une image arbitraire par un ensemble fini de pixels (10×10 dans cet exemple). Pour une image en noir et blanc, cet ensemble de pixels se traduit directement en une matrice de bits, 0 pour noir et 1 pour blanc par exemple. On parle donc de représentation *bitmap* des images.

Si on veut introduire plus de nuances, on peut commencer par attribuer à chaque pixel non plus la couleur noir ou blanc, mais un *niveau de gris* représenté par un entier entre 0 (pour noir) et 255 (pour blanc). On choisit cette échelle car un pixel se représente alors exactement sur un octet et cela suffit à exprimer des nuances quasi indiscernables à l'œil nu. On verra en fin de TP comment cette représentation s'étend à des pixels colorés.

Les images seront lues et écrites dans des fichiers au moyen des commandes présentées en annexe B.2. On travaille pour l'instant en niveaux de gris : si un objet `im` de type `Image` est en couleurs, on peut le convertir en niveaux de gris via la méthode `im.convert('L')`.

On suppose à partir de maintenant qu'on travaille sur le tableau `numpy.array` qui contient les valeurs des pixels entre 0 et 255. On ne mentionne pas les fonctions d'ouverture, de visualisation et d'enregistrement d'images.

A.7.1 Traitement pixel par pixel

On commence par une opération simple : effacer l'image, autrement dit rendre tous ses pixels blancs :

```
def effacer(image):
    for ligne in image:
        for i in range(len(ligne)):
            ligne[i] = 255
```

On peut aussi vouloir créer le négatif (au sens photographique) de l'image. Cela consiste à inverser le blanc et le noir, le gris clair et le gris foncé, etc. Une simple fonction affine de coefficient directeur négatif appliquée à la valeur des pixels fait l'affaire :

```
def negatif(image):
    for ligne in image:
        for i in range(len(ligne)):
            ligne[i] = 255 - ligne[i]
```

Il apparaît vite que beaucoup des fonctions de traitement de l'image auront cette structure. On peut donc commencer par rendre leur écriture plus efficace.

1 Écrire une fonction `traitement` qui prend pour arguments une image et une fonction, puis applique cette fonction sur la valeur de chacun des pixels de l'image.

2 Redéfinir `effacer` et `negatif` à l'aide de `traitement`. L'utilisation d'une fonction anonyme est recommandée.

3 On joue maintenant sur la luminosité de l'image : puisque les valeurs les plus hautes sont les plus claires, il suffit d'augmenter les valeurs des pixels pour éclaircir l'image et de les diminuer pour l'assombrir. On est alors tenté d'appeler `traitement` avec une fonction comme `lambda x: x + 50`.

   On s'aperçoit vite que l'effet obtenu n'est pas celui escompté : tous les pixels dont la valeur était supérieure à 205 reçoivent une valeur entre 255 et 305. Comme les valeurs des pixels sont comptées modulo 255, ces pixels deviennent en fait presque noirs.

   Écrire une fonction qui éclaircit une image en s'assurant par un test que la valeur des pixels est plafonnée à 255.

4 La solution précédente n'est pas satisfaisante car elle crée de grands aplats blancs dans les zones les plus claires (on pourra augmenter la valeur des pixels de 100 ou 150 pour s'en convaincre).

   À l'aide de la fonction racine carrée, construire une bijection de [0; 255] dans [0; 255]. Programmer cette fonction pour qu'elle manipule et renvoie des entiers (ce n'est alors évidemment plus une bijection).

   Passer cette fonction en paramètre à `traitement` et vérifier le résultat obtenu[9].

5 Construire de même une solution pour assombrir une image.

6 Tracer le graphe de la fonction $f : x \mapsto 128 + signe(x - 128) \times \sqrt{128\,|x - 128|}$ sur [0; 255].

   À l'aide de ce graphe, prévoir l'effet qu'aura cette fonction sur l'image, puis tester votre hypothèse.

7 Expérimenter avec des fonctions linéaires, non monotones, etc.

A.7.2 Traitement local

On s'intéresse maintenant à des traitements qui font intervenir plusieurs pixels voisins. La fonction `traitement` écrite dans la partie précédente n'est donc plus utilisable.

1 Écrire une fonction qui produit une image floutée. Pour cela, il suffit de remplacer chaque pixel par la moyenne des pixels qui l'entourent.

9. Cette méthode se généralise aux fonctions de la forme $x \mapsto x^\gamma$ avec $\gamma > 0$. C'est la fameuse *correction gamma* proposée dans les logiciels de retouche photographique.

On obtient différents degrés de floutage selon qu'on considère les 9 pixels immédiatement adjacents, ou bien un carré de 5 voire 7 pixels de côté centré sur le pixel à calculer.

2 À l'inverse, on peut chercher des contours dans une image ; il s'agit de détecter les endroits où les pixels changent brutalement de couleur. Pour cela, on calcule pour chaque pixel une moyenne pondérée des pixels qui l'entourent, par exemple avec les coefficients :

-1	-1	-1
-1	8	-1
-1	-1	-1

Quel est le résultat de ce calcul pour un pixel qui est entouré d'autres pixels d'une couleur proche ? Et si au contraire il est d'une couleur nettement différente de celle d'une partie de ses voisins ?

Programmer cette méthode de détection des contours.

3 Expérimenter avec d'autres coefficients. On retrouve ainsi encore d'autres fonctionnalités des logiciels de traitement d'image.

A.7.3 Traitement global

On peut enfin chercher à modifier l'image dans sa globalité.

1 Écrire une fonction qui agrandit une image d'un facteur $k > 1$ donné. Pour k entier, il suffit de remplacer chaque pixel par un bloc de $k \times k$ pixels. Si k est décimal, il faudra procéder à des interpolations pour calculer la couleur des pixels de l'image agrandie.

2 Écrire une fonction qui divise par deux les dimensions d'une image. Chaque bloc de 2×2 pixels sera donc remplacé par un unique pixel dont la valeur est la moyenne des quatre pixels originaux.

3 Généraliser ce principe pour écrire une fonction qui réduit la taille d'une image d'un facteur $k < 1$.

4 Que se passe-t-il si on prend deux images de même taille et qu'on calcule leur différence pixel par pixel ?

A.7.4 En couleurs

Pour représenter une couleur, on peut la décomposer en trois intensités de bleu, de vert et de rouge ; c'est d'ailleurs la façon dont l'œil humain perçoit les couleurs. Numériquement, cela signifie qu'on peut conserver le même principe de représentation, en donnant pour chaque pixel trois nombres entre 0 et 255. Il faut donc trois octets pour représenter un pixel de couleur.

Dans les matrices créées par le module `Image` pour représenter des images en couleur, cela se traduit par le fait que chaque pixel est représenté non plus par un entier mais par un tableau de trois entiers.

Les algorithmes écrits aux questions précédentes restent pour la plupart valables sur des images en couleur, en les appliquant composante par composante.

Il est également possible d'appliquer un traitement différent sur chaque composante, par exemple accentuer le niveau de rouge et atténuer celui de bleu et de vert pour obtenir des couleurs plus chaudes.

A.8 Prise en main de phpMyAdmin

Pour créer une base de données, il est souvent plus simple d'utiliser une interface graphique que d'écrire des requêtes SQL. Dans cet ouvrage, on n'a d'ailleurs présenté que des requêtes de recherche, pas de création ni de modification d'une base. Dans ce TP, on présente un tour d'horizon de l'interface phpMyAdmin.

On considère ici que phpMyAdmin a déjà été installé et est fonctionnel. Un utilisateur possédant une base de données appelée tpbdd a déjà été créé.

On adopte le vocabulaire des bases de données, dont le tableau suivant présente les principales correspondances avec le modèle relationnel :

Modèle relationnel	Bases de données
relation	table
attribut	colonne
valeur	ligne
domaine	type

A.8.1 Création d'une table

En accédant la première fois à l'adresse HTTP où phpMyAdmin est installé, l'utilisateur est invité à s'authentifier. Après avoir entré son identifiant et son mot de passe, il est redirigé vers l'écran d'accueil (figure A.22).

La partie gauche de l'interface sert à parcourir les bases de données auxquelles on a accès. Ici, seule la base tpbdd est disponible. En cliquant dessus, on est dirigé sur l'interface de manipulation des tables de la base de données. Dans le cas présent, il n'y en a pas et l'interface invite à en créer une (figure A.23).

On choisit de créer une table nommée *livre* et contenant deux colonnes.

Après avoir appuyé sur *Exécuter*, on découvre l'interface de définition des colonnes (figure A.24). Celle-ci a été préinitialisée pour proposer deux colonnes, car c'est le nombre qui avait été annoncé dans la boîte de dialogue précédente.

La première chose que l'on remarque ici, c'est que l'interface propose bien plus de possibilités que la simple définition d'un nom et d'un type. En effet, le modèle relationnel est épuré par rapport aux gestionnaires de bases de données.

Pour le moment, on ne remplit que le nom et le type des colonnes et on laisse les autres champs avec leur valeur par défaut.

Pour choisir le type d'une colonne, l'interface propose un menu déroulant avec un choix intimidant (figure A.25). Voici un descriptif des principaux types :

Tableau A.1 Principaux types proposés pour les colonnes

`INT`	Le type des entiers relatifs. La plus grande et la plus petite valeur pouvant être stockées dépendent du système (voir chapitre 2).
`TEXT`	Le type des chaînes de caractères de longueur quelconque. Ce type est difficile à manipuler par le gestionnaire de bases de données et on lui préférera le type suivant dans la majorité des cas.
`VARCHAR`	Le type des chaînes de caractères de longueur maximale fixée (espaces compris). Cette longueur est fournie dans le champ suivant de l'interface. Cette contrainte aide le gestionnaire de bases de données à gérer au mieux ces valeurs.
`DECIMAL`	Le type des nombres décimaux. C'est la représentation à privilégier pour des données financières, car présentant le moins d'erreurs d'arrondis.
`FLOAT`	Le type des nombres à virgule flottante (voir chapitre 2).

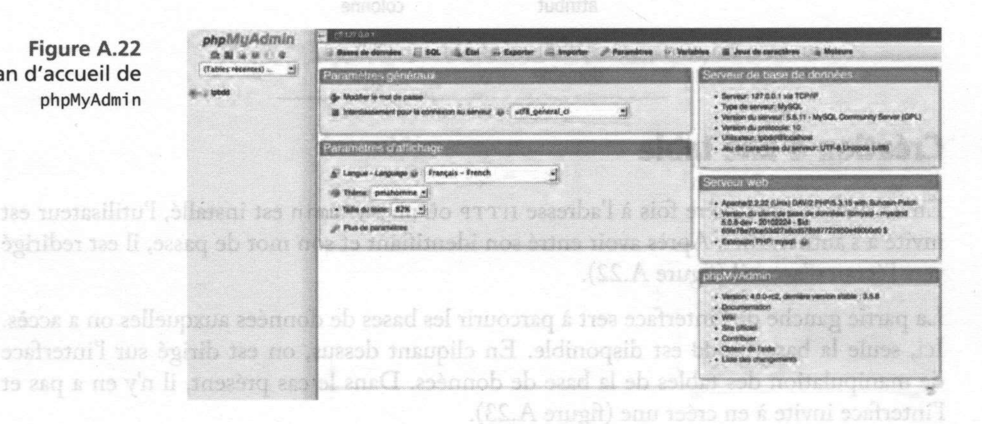


Figure A.22 L'écran d'accueil de `phpMyAdmin`

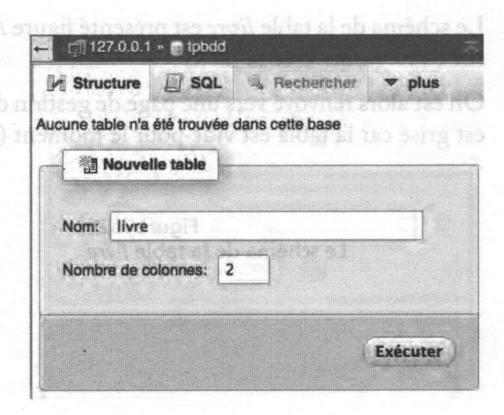


Figure A.23
Création d'une nouvelle table

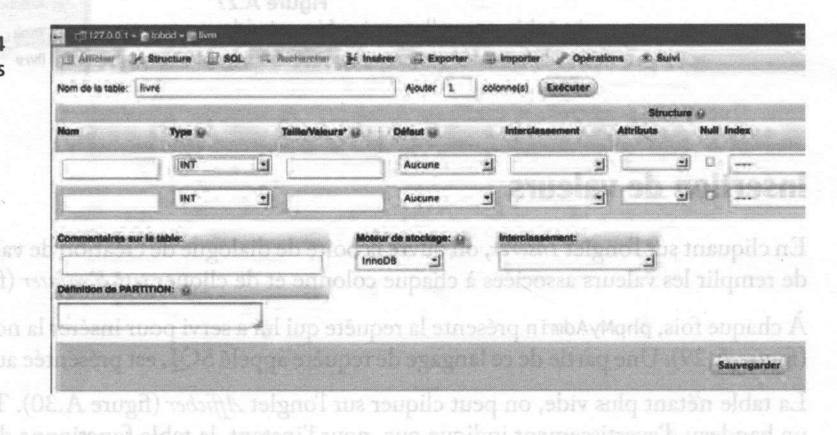


Figure A.24
Définition des colonnes

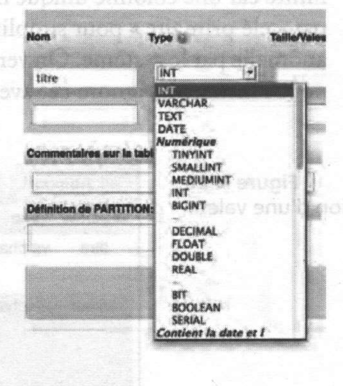
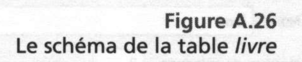


Figure A.25
Types des colonnes

Le schéma de la table *livre* est présenté figure A.26. Une fois ces champs remplis, un appui sur *Sauvegarder* crée la table.

On est alors renvoyé vers une page de gestion de la table. On remarque que l'onglet *Afficher* est grisé car la table est vide pour le moment (figure A.27).



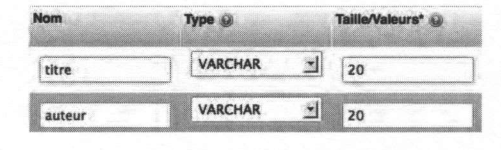


Figure A.26
Le schéma de la table *livre*

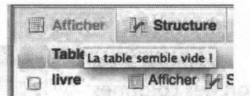


Figure A.27
La table nouvellement créée est vide.

A.8.2 Insertion de valeurs

En cliquant sur l'onglet *Insérer*, on ouvre la boîte de dialogue de création de valeurs. Il suffit de remplir les valeurs associées à chaque colonne et de cliquer sur *Exécuter* (figure A.28).

À chaque fois, `phpMyAdmin` présente la requête qui lui a servi pour insérer la nouvelle valeur (figure A.29). Une partie de ce langage de requête appelé SQL est présentée au chapitre 11.

La table n'étant plus vide, on peut cliquer sur l'onglet *Afficher* (figure A.30). Tout d'abord, un bandeau d'avertissement indique que, pour l'instant, la table fonctionne dans un mode limité car une colonne unique n'a pas été définie. Ici, on peut remplacer « colonne unique » par « clé primaire » pour simplifier : une table sans clé primaire est considérée comme une anomalie par le système. On verra dans le prochain TP comment définir des clés primaires ; pour l'instant on ignore cet avertissement.

Figure A.28
Insertion d'une valeur

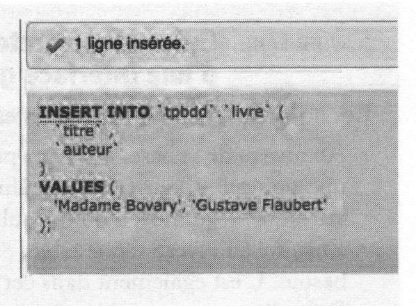


Figure A.29
Après l'insertion

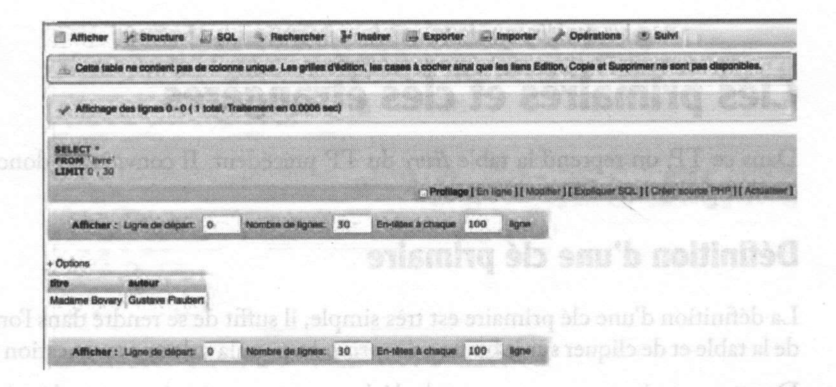


Figure A.30
Affichage de la table

1 Insérer des valeurs pour obtenir l'affichage de la table correspondant à la figure A.31.
2 Créer une nouvelle table *vol* modélisant les liaisons aériennes au départ d'un aéroport. Chaque vol aura pour attributs un numéro, un horaire et une destination.
3 Créer une nouvelle table *voyageur* modélisant les voyageurs qui se présentent à cet aéroport, avec au moins quatre attributs bien choisis.
4 Insérer quelques valeurs représentatives dans la table des vols et dans la table des voyageurs.

Figure A.31
Affichage de la table *livre* remplie

titre	auteur
Madame Bovary	Gustave Flaubert
Salammbô	Gustave Flaubert
Le rouge et le noir	Stendhal
Germinal	Emile Zola

Savoir-faire Créer et alimenter une base de données simple à l'aide d'une interface graphique

Les noms des menus sont explicites :

- Au niveau de la base, *Structure* permet d'accéder aux tables existantes et d'en créer une nouvelle (pour cela, on définit son schéma relationnel — noms et types des attributs — au moyen du formulaire proposé).
- Ensuite, au niveau d'une table, *Structure* sert à modifier son schéma relationnel au besoin. C'est également dans cet onglet qu'on peut déclarer une clé primaire.
- Pour alimenter une table, on clique sur *Insérer* et on remplit un formulaire.

A.9 Clés primaires et clés étrangères

Dans ce TP, on reprend la table *livre* du TP précédent. Il conviendra donc de la créer si ce n'est pas encore fait.

A.9.1 Définition d'une clé primaire

La définition d'une clé primaire est très simple, il suffit de se rendre dans l'onglet *Structure* de la table et de cliquer sur le bouton *Clé primaire* pour la colonne en question (figure A.32).

Dans un premier temps, on essaye de déclarer **auteur** comme étant une clé primaire de la relation *livre*. Une boîte de dialogue présente la commande qui sera effectivement transmise au gestionnaire de base de données (figure A.33). Après avoir appuyé sur *OK*, un message d'erreur indique que la valeur "Gustave Flaubert" est dupliquée (figure A.34). Ainsi, la colonne **auteur** n'est pas une clé primaire. Cela signifie que le gestionnaire de bases de données ne se contente pas de considérer n'importe quelle colonne comme une clé primaire ; il vérifie que c'est le cas.

La colonne **titre** est ici une clé primaire ; on reprend donc les étapes précédentes avec cette colonne. L'ajout s'effectue sans erreur et on voit dans l'arborescence des tables à gauche de la fenêtre, qu'une entrée `Index` a été ajoutée et qu'elle contient la clé primaire (figure A.35). Dans l'onglet *Structure*, il est aussi possible de cliquer sur le lien *Index* pour voir la clé primaire et la manipuler (figure A.36). L'encadré ci-après fournit des compléments sur cette notion d'index.

#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
1	titre	varchar(20)	latin1_swedish_ci		Non	*Aucune*		Modifier Supprimer Primaire
2	auteur	varchar(20)	latin1_swedish_ci		Non	*Aucune*		Modifier Supprimer Primaire

Figure A.32
Bouton *Clé primaire*

Figure A.33
Confirmation de la déclaration en tant que clé primaire

Voulez-vous vraiment exécuter «ALTER TABLE `livre` ADD PRIMARY KEY(`auteur`);» ?

OK Annuler

Figure A.34
Erreur car la clé primaire est invalide

Erreur dans le traitement de la requête :

#1062 - Duplicate entry 'Gustave Flaubert' for key 'PRIMARY'

Figure A.35
Arborescence de la base

tpbdd
Nouvelle table
livre
Colonnes
Nouvelle colonne
auteur
titre
Index
Nouvel index
PRIMARY

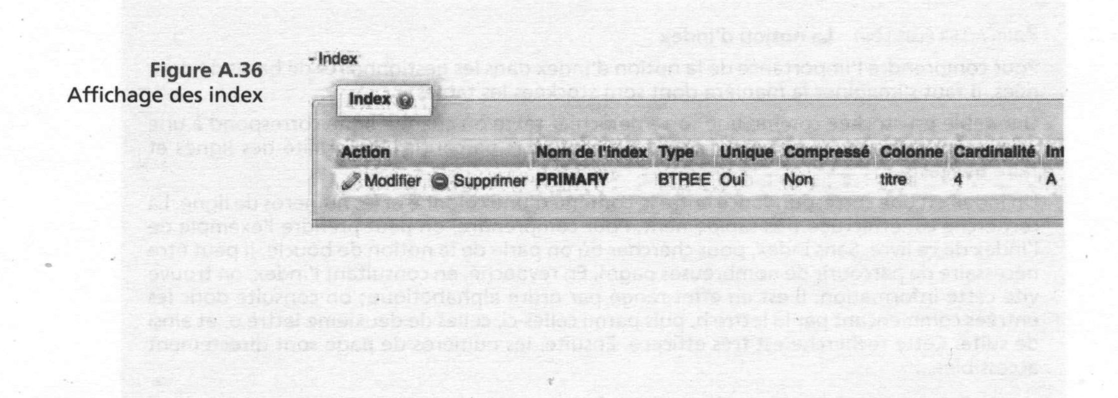


Figure A.36
Affichage des index

Si on essaye d'effectuer une insertion avec le titre d'un livre existant (figure A.37), le gestionnaire renvoie une erreur (figure A.38). Ainsi, le système s'assure que la colonne reste une clé primaire à chaque insertion ou modification.

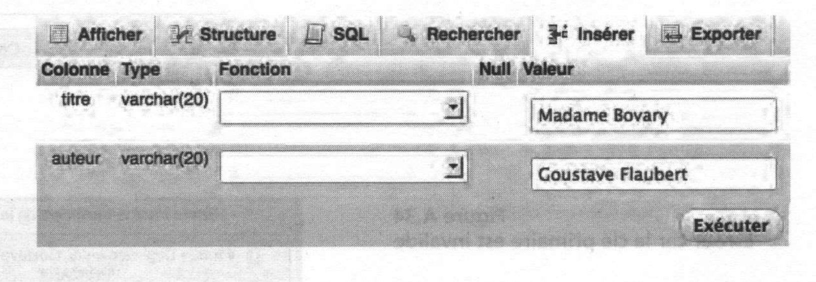


Figure A.37
Insertion avec un titre existant

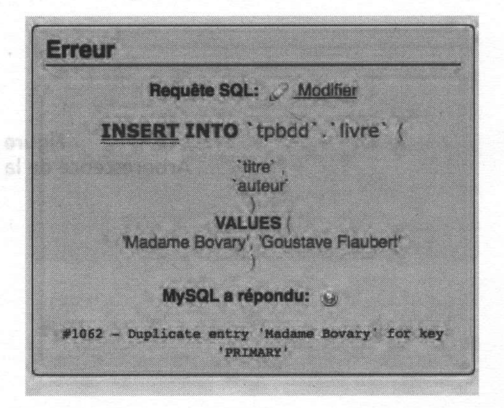


Figure A.38
Erreur de violation de la clé primaire

POUR ALLER PLUS LOIN **La notion d'index**

Pour comprendre l'importance de la notion d'index dans les gestionnaires de bases de données, il faut s'imaginer la manière dont sont stockées les tables.

Une table est stockée comme une sorte de fichier texte où chaque ligne correspond à une valeur. Pour chercher une valeur précise, il faut alors parcourir l'intégralité des lignes et faire des tests.

Un index est une correspondance entre le contenu d'une colonne et les numéros de ligne. La recherche est effectuée très rapidement. Pour comprendre, on peut prendre l'exemple de l'index de ce livre. Sans index, pour chercher où on parle de la notion de boucle, il peut être nécessaire de parcourir de nombreuses pages. En revanche, en consultant l'index, on trouve vite cette information. Il est en effet rangé par ordre alphabétique ; on consulte donc les entrées commençant par la lettre b, puis parmi celles-ci, celles de deuxième lettre o, et ainsi de suite. Cette recherche est très efficace. Ensuite, les numéros de page sont directement accessibles.

MySQL propose trois types d'index :

- **Les clés primaires.** Elles identifient uniquement les valeurs d'une table. On ne peut définir qu'une seule clé primaire par table.
- **Les index uniques.** Il s'agit de clés primaires optionnelles. Une colonne peut être un index unique si toutes ses valeurs différentes de NULL sont uniques. Une table peut comporter plusieurs index uniques.
- **Les index.** Il s'agit de simples correspondances sans condition d'unicité. C'est le cas de l'index de ce livre : une notion peut être associée à plusieurs pages.

Les index accélèrent les recherches, mais ralentissent les insertions et les modifications. Ainsi, il n'est pas raisonnable d'indexer toutes les colonnes, de même que dans l'index d'un livre, il n'est pas judicieux de mentionner tous les termes utilisés dans l'ouvrage. Choisir les bonnes colonnes à indexer dépend d'une analyse fine des besoins.

A.9.2 Clé primaire auto-incrémentée

MySQL, comme beaucoup d'autres gestionnaires de bases de données, dispose d'un mécanisme de création automatique de clé primaire. Il s'agit d'attribuer à chaque valeur un identifiant numérique unique et, lors d'une insertion, d'affecter le numéro suivant. On parle de clé primaire *auto-incrémentée.*

Comme il n'est pas possible d'avoir deux clés primaires pour une même table, on commence par supprimer l'ancienne clé primaire. Dans la boîte de dialogue de la figure A.36, on clique sur *Supprimer* et on confirme.

On ajoute maintenant une nouvelle colonne en début de table dans l'onglet *Structure* (figure A.39).

Pour ajouter une clé primaire auto-incrémentée, il faut définir une colonne de type INT. Dans la liste *Index*, on sélectionne PRIMARY et on coche la case A.I., pour AUTO_INCREMENT (figure A.40).

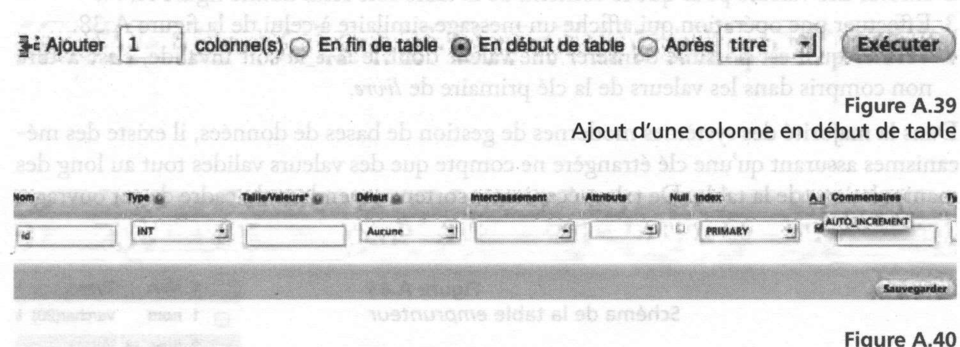


Figure A.39
Ajout d'une colonne en début de table

Figure A.40
Champ pour la création d'une clé primaire auto-incrémentée

On peut alors constater dans l'onglet *Structure* qu'une nouvelle colonne a été ajoutée en début de table et, dans le champ EXTRA, on peut lire AUTO_INCREMENT (figure A.41).

En cliquant sur l'onglet *Afficher*, on voit que les lignes de cette table s'affichent dans l'ordre croissant de la nouvelle clé primaire (figure A.42).

	#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra
☐	1	id	int(11)			Non	*Aucune*	AUTO_INCREMENT
☐	2	titre	varchar(20)	latin1_swedish_ci		Non	*Aucune*	
☐	3	auteur	varchar(20)	latin1_swedish_ci		Non	*Aucune*	

Figure A.41
La nouvelle structure de la table *livre* avec la colonne id

Figure A.42
Valeurs de la table *livre* modifiées par l'ajout de la colonne id

id	titre	auteur
1	Germinal	Emile Zola
2	Le rouge et le noir	Stendhal
3	Madame Bovary	Gustave Flaubert
4	Salammbô	Gustave Flaubert

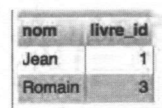
A.9.3 Lien entre deux tables

1 Créer une table *emprunteur* dont le schéma est donné figure A.43. La colonne livre_id représente une clé étrangère vers la table *livre* munie de la clé primaire id.
2 Insérer des valeurs pour que le contenu de la table soit celui donné figure A.44.
3 Effectuer une opération qui affiche un message similaire à celui de la figure A.38.
4 Vérifier qu'il est possible d'insérer une valeur dont le livre_id soit invalide, c'est-à-dire non compris dans les valeurs de la clé primaire de *livre*.

Dans la majorité des systèmes modernes de gestion de bases de données, il existe des mécanismes assurant qu'une clé étrangère ne compte que des valeurs valides tout au long des manipulations de la table. De tels mécanismes sortent cependant du cadre de cet ouvrage.

Figure A.43
Schéma de la table *emprunteur*

	#	Nom	Type
☐	1	nom	varchar(20)
☐	2	livre_id	int(11)



nom	livre_id
Jean	1
Romain	3

Figure A.44
Contenu de la table *emprunteur*

A.9.4 Lancement de requêtes

Pour lancer une requête sur une base, il suffit d'aller dans l'onglet *SQL* et d'y taper la requête en langage SQL avec la syntaxe présentée au chapitre 11. Le résultat de la requête s'affiche sous forme de table ; si on s'aperçoit qu'on a commis une erreur dans la requête, il est possible de la corriger puis de la relancer.

L'onglet *Rechercher* permet par ailleurs d'effectuer des projections et des sélections élaborées sans avoir à écrire de conditions booléennes, mais il se limite à ce type de requête.

1 Reprendre (ou créer) les tables *vol* et *voyageur*. Créer une nouvelle table *avion* avec une clé primaire auto-incrémentée modélisant la flotte d'avions qui transitent par cet aéroport.
2 De quelle table faut-il modifier le schéma relationnel pour affecter un avion à chaque vol ? Le faire.
3 Écrire une requête qui détermine la destination de chaque avion.
4 Écrire une requête qui détermine l'identifiant de l'avion que prendra chaque passager.
5 Écrire une requête qui compte combien de passagers embarqueront sur chaque vol.
6 Écrire une requête qui vérifie que le même avion n'est pas affecté à deux vols distincts.

B

Compléments sur les entrées/sorties

Cette annexe décrit trois façons de produire des programmes qui utilisent ou produisent des données volumineuses ou non numériques.

B.1 Lecture et écriture dans des fichiers

On présente ici le « kit de survie » pour qui veut lire et écrire dans des fichiers en Python. Une fois le principe compris, l'utilisation est extrêmement simple : c'est un point fort de Python.

B.1.1 Lire les lignes d'un fichier

Le principe est le suivant :

1 On ouvre un fichier.
2 On lit les lignes successivement.
3 On ferme le fichier.

En pratique :

- L'ouverture du fichier consiste à associer (via la fonction **open**) un objet de type *file* à un fichier existant. On peut voir cet objet comme quelque chose qui pointe au début du fichier et auquel on peut demander de lire des lignes et d'en donner le résultat. *Le principe n'est pas d'aller voir à un endroit arbitraire, contrairement à ce que l'on ferait dans un tableau.*
- La lecture des lignes peut se faire soit pas à pas via la méthode **readline**, qui lit la ligne courante et passe à la suivante, soit globalement via la méthode **readlines**, qui crée l'itérateur de toutes les lignes et permet ainsi de traiter ces dernières dans une boucle. Chaque ligne est en fait une chaîne de caractères (*string*).
- On ferme le fichier via la méthode **close** : il est conseillé d'écrire cette instruction aussitôt après avoir écrit l'instruction **open**. Si on l'oublie, la plupart du temps on ne verra pas la différence. Néanmoins, c'est une bonne habitude à prendre car on libère ainsi des ressources dont on n'a plus l'utilité.

Dans l'exemple suivant, on ouvre un fichier dont le nom est `tagada.txt`, on affiche la première ligne lue, puis on calcule la somme des longueurs de toutes les lignes suivantes.

```
mon_fichier = open('tagada.txt','r') # 'r' pour lecture
print(mon_fichier.readline())
s = 0
for L in mon_fichier.readlines():
    s += len(L)
mon_fichier.close()
```

Une façon encore plus simple de parcourir les lignes d'un fichier consiste à écrire : **for** *L* **in** *mon_fichier*. Cela ressemble à la méthode **readlines**, à une subtile amélioration près. La séquence des lignes, qui peut être grande, n'est pas créée avant d'être parcourue : les lignes sont lues à la volée, dès que la précédente a été traitée.

Exercice B.1 avec corrigé Exécuter le programme précédent sur un petit fichier de quelques lignes que vous aurez préalablement créé. Comparer le résultat avec celui attendu, puis expliquer.

Chaque ligne `L` *est terminée par un caractère spécial de* retour à la ligne *que l'on note* `'n'`*. Il est affiché par* **print** *et compté par* **len***. On peut le visualiser en observant la valeur de* `L` *ou bien en la transformant en liste.*

B.1.2 Extraction des données dans une ligne

En pratique, on veut souvent séparer des informations dans les lignes lues. Typiquement, une ligne prend la forme *Sharapova,2,Russe* et on veut en extraire différentes données, de différentes natures. La méthode **split** fait une partie de ce travail : il suffit de lui donner comme paramètre le caractère selon lequel on veut couper la chaîne. Une liste de chaînes est rendue :

```
In [1]: 'Sharapova,2,Russie'.split(',')
Out[1]: ['Sharapova', '2', 'Russie']
```

Si un fichier regroupant quelques lignes de ce type a été écrit, la lecture va réserver une mauvaise surprise :

```
In [2]: f = open('tennis.txt')
```

```
In [3]: f.readline()

Out[3]: 'Sharapova,2,Russie\n'
```

Pour enlever les espaces et caractères d'échappement (ici, un retour à la ligne) en début et fin de ligne, on dispose de la méthode **strip**, déclinable à droite et à gauche (resp. **rstrip** et **lstrip**) :

```
In [4]: '\n \t toto \t truc \n'.strip()

Out[4]: 'toto \t truc'
```

Dans l'exemple suivant, on lit un fichier contenant des noms, classements et nationalités de joueuses de tennis :

```
f = open('tennis.txt','r')

for L in f:
    nom, classement, nation = L.rstrip().split(',')
    print(nom, nation, int(classement))

f.close()
```

Le résultat est :

```
('Sharapova', 'Russie', 2)
('Williams', 'USA', 1)
...
```

> **Attention** **Déconstruction des listes**
> On aura noté l'affectation de la forme `a, b, c = [1, 2, 3]` qui fait ce qu'on peut raisonnablement espérer, même s'il conviendrait de l'écrire de façon plus homogène : `a, b, c = 1, 2, 3` ou éventuellement `[a, b, c] = [1, 2, 3]`.
> C'est ce type de souplesse qui fait que certains programmeurs trouvent Python formidable... quand d'autres le trouvent laxiste.

B.1.3 Écrire des données dans un fichier

Pour écrire dans un fichier, on commence par l'ouvrir *en écriture* (ce qui aura pour effet d'écraser ledit fichier, s'il existait déjà [1]). Ensuite, on y écrit des chaînes de caractères via la méthode **write** et on termine en fermant le fichier. Attention, ici la fermeture est cruciale : c'est à cet instant que le fichier va réellement être écrit sur le disque.

1. On peut choisir d'écrire à la fin du fichier préexistant, en prenant comme option 'a' (pour « *append* ») plutôt que 'w'.

Dans l'exemple suivant, on reprend le fichier contenant des données sur des joueuses de tennis, on récupère la liste des lignes et on ferme le fichier aussitôt après. On ouvre ensuite un nouveau fichier dans lequel on écrit les mêmes données, mais formatées différemment (les données sont séparées par des tabulations) :

```
f = open('tennis.txt','r')
liste = f.readlines()
f.close()

f = open('tennis-bis.txt','w')
for joueuse in liste: # joueuse est une chaine de caractères
    tab = joueuse.strip().split(',') # tab est un tableau
    f.write(tab[0]+'\t'+tab[1]+'\t'+tab[2]+'\n')
f.close()
```

Exercice B.2 Si *c* est une chaîne de caractères, `c.upper()` renvoie une chaîne où toutes les minuscules de *c* ont été remplacées par des majuscules :

```
In [5]: 'Hop'.upper()

Out[5]: 'HOP'
```

Écrire une suite d'instructions copiant un fichier en changeant au passage toutes les minuscules en majuscules.

On pourra appliquer ceci au fichier `.py` *contenant ces instructions !*

Exercice B.3 Un fichier texte contient 5 398 lignes de la forme :

```
1234  TOURNESOL Tryphon    Admissible   2
```

ou bien :

```
43210 HADDOCK Archibald    Eliminé      -
```

(Des tabulations séparent les quatre champs ; le nom et le prénom séparés par un espace sont dans un même champ.)

Écrire une suite d'instructions qui lit ce fichier, détermine le nombre d'admissibles et le nombre de candidats attribués à chaque série d'oral.

Une recherche sur Internet sur « admissibles mines 2013 » suivie d'un copier/coller de texte doit permettre au lecteur de ne pas travailler dans l'abstraction.

Exercice B.4 On s'intéresse maintenant à l'écriture et à la lecture de matrices d'entiers sous différents formats. Dans chaque cas, il est demandé d'écrire une fonction écrivant (resp. lisant) une matrice dans un fichier.

1 On commence par écrire les dimensions de la matrice sur la première ligne (`4*12`), puis on écrit une autre ligne contenant toutes les données séparées par des virgules (en commençant par les éléments de la première ligne par exemple).
2 On écrit les dimensions sur la première ligne, puis chaque ligne de la matrice est écrite dans une ligne du fichier.
3 On se contente d'écrire les lignes de la matrice sur des lignes différentes du fichier, en séparant les éléments d'une ligne par des tabulations.
4 On écrit la matrice sur une seule ligne, au format `[[a(1,1),...,a(1,n)],...,[a(m,1),...,a(m,n)]]`.

Comment modifier les différentes fonctions pour autoriser la manipulation de matrices de flottants ?

Exercice B.5 On trouve sans trop de mal le texte original de *Hamlet* sur Internet.

1 Récupérer et enregistrer la version française de ce texte dans un fichier `.txt`.
2 À l'aide de Python, compter le nombre d'interventions des différents protagonistes. Faire des statistiques sur le nombre de mots, de lettres, lesquels sont le plus utilisés, etc.
3 Recommencer avec la version originale.

B.2 Lecture et écriture dans des images

Le module **Image** fournit un ensemble de fonctions pour ouvrir un fichier d'image, en récupérer le contenu sous un format manipulable par un programme Python et enregistrer le résultat dans un nouveau fichier image. Il ne fait pas partie de la bibliothèque standard, mais il est souvent inclus dans les distributions Python.

B.2.1 Lecture d'image

Comme pour les autres fichiers, on commence par ouvrir l'image via une fonction **open**, mais il s'agit ici de la fonction de la bibliothèque **Image** :

```
In [6]: mb = Image.open('mario.png')
```

À partir de là, *mb* ne désigne pas l'image stockée sur le disque dur : il s'agit d'un objet abstrait que l'on manipule dans le programme et qu'on pourra plus tard éventuellement enregistrer dans un fichier.

- On peut afficher l'image à l'aide de la méthode **show** :

```
In [7]: mb.show()
```

- Ses dimensions sont données par l'attribut *size* (sans parenthèses, il ne s'agit pas d'une méthode) :

```
In [8]: mb.size
Out[8]: (201, 251)
```

- L'attribut *mode* vaut souvent `'RGB'` pour les images en couleur et `'L'` pour celles en niveaux de gris.

B.2.2 Traitement

Avant d'effectuer un traitement sur l'image, il faut décomposer cette dernière en pixels. Pour cela, avec le module **numpy**, on transforme directement l'image en tableau :

```
In [9]: mbtab = numpy.array(mb)
```

Chaque pixel est représenté par un entier entre 0 et 255 pour une image en niveaux de gris, ou par un tableau de trois entiers entre 0 et 255 pour une image en couleurs. Le TP A.7 donne des informations plus approfondies sur les raisons de cette représentation et sur les manipulations qu'elle permet.

On se contente ici d'appliquer un traitement simple sur les couleurs :

```
for lignes in mbtab:
    for m in lignes:
        m[0],m[1] = m[1],m[0]
```

B.2.3 Écriture dans une image

On dispose ensuite d'une fonction qui permet de créer une image à partir du tableau de ses pixels :

```
In [10]: mbnew = Image.fromarray(mbtab)
```

Enfin, pour enregistrer une image, on utilise la méthode `save`. L'extension donnée au nom du fichier détermine le format d'enregistrement, qui n'est pas forcément le même que celui de l'image d'origine :

```
In [11]: mbnew.save('luigi.jpg')
```

On consultera bien entendu la documentation en ligne de `Image` pour découvrir d'autres fonctions.

Exercice B.6 Écrire une fonction qui ouvre une image et compte le nombre de pixels parfaitement noirs et de pixels parfaitement blancs dans celle-ci.

Exercice B.7 Écrire une fonction qui ouvre une image et en enregistre une copie agrandie d'un facteur 2, en hauteur comme en largeur. Chaque pixel de l'image originelle est donc remplacé par quatre pixels identiques.

Exercice B.8 Écrire une fonction qui ouvre une image et affiche l'image symétrique par rapport à un axe vertical.
Même question en faisant subir cette fois à l'image une rotation de 90°.

Exercice B.9 Écrire une fonction qui prend pour arguments deux entiers n et p et crée l'image d'un damier de dimensions $n \times p$: les pixels sont alternativement noirs et blancs le long des lignes comme le long des colonnes.

B.3 Utilisation du module graphique `turtle`

Le module `turtle` sert à réaliser des tracés géométriques à l'aide d'un curseur appelé *tortue*, comme dans le célèbre langage Logo mis au point en 1967 par Wally Feurzeig et Seymour Papert pour la première utilisation d'ordinateurs à des fins éducatives.

La tortue est symbolisée par une flèche, dispose d'un crayon et d'une « tête » qui fixe sa direction. Partout où elle passe, la tortue trace un trait si son crayon est baissé. Elle effectue les tracés suffisamment lentement pour que le programmeur puisse suivre la suite d'instructions au fur et à mesure qu'elles s'exécutent. On dispose d'un ensemble d'instructions élémentaires pour déplacer la tortue ou tourner sa tête :

Les fonctions du module **turtle** (tortue Logo)	
reset()	Effacer le dessin
goto(*x*, *y*)	Aller au point de coordonnées x, y sans tourner la tête
forward(*distance*)	Avancer d'une distance donnée
backward(*distance*)	Reculer d'une distance donnée
up()	Lever le crayon (pour ne plus dessiner)
down()	Baisser le crayon (pour dessiner)
left(*angle*)	Tourner la tête à gauche d'un angle exprimé en degrés
right(*angle*)	Tourner la tête à droite d'un angle exprimé en degrés

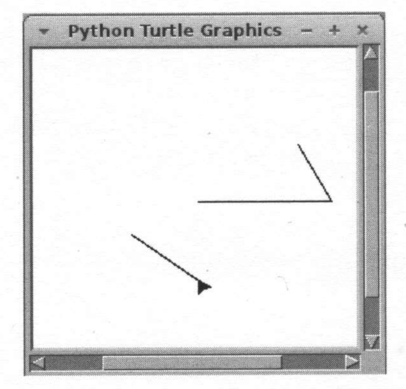
Toutes ces instructions sont à précéder du nom du module, qu'on aura donc intérêt à charger sous la forme `import turtle as t`. On peut aussi taper `from turtle import *` pour utiliser ces instructions directement sans préfixe. Des fonctions plus avancées sont disponibles, que l'on retrouvera sur la documentation en ligne du module.

Pour que la fenêtre de tracé ne se bloque pas en fin d'exécution, il faut terminer le programme par l'instruction `t.mainloop()`.

Le programme suivant produit le dessin figure B.1.

```
import turtle as t
t.forward(100)
t.left(120)
t.forward(50)
t.up()
t.goto(-50,-25)
t.down()
t.goto(0,-60)
t.mainloop()
```

Figure B.1
Un tracé à l'aide du module **turtle**



Exercice B.10 Écrire un programme qui trace une lettre H sans utiliser la fonction `up()`.

Exercice B.11 Écrire une fonction qui trace en pointillés une ligne de longueur a donnée en argument dans la direction pointée par la tortue.

Adapter la fonction pour que la longueur des pointillés puisse être aussi donnée en argument.

Exercice B.12 Écrire une fonction qui trace le segment reliant les points de coordonnées (x_1, y_1) et (x_2, y_2) fournies en arguments, sans effectuer d'autre tracé et quelle que soit la position initiale de la tortue.

Exercice B.13 Écrire une fonction qui trace un carré de côté a donné en argument. On considère qu'au moment de l'appel la tortue est placée sur un des sommets du carré et regarde dans la direction d'un des côtés.

Même question pour un triangle de côté donné.

Généraliser cette fonction pour qu'elle prenne en argument le nombre de côtés du polygone à tracer.

Références

Système

A. S. Tanenbaum. *Systèmes d'exploitation*, Pearson, 2008.

Représentation des nombres

J.-C. Bajard et J.-M. Muller (coordinateurs). *Calcul et arithmétique des ordinateurs*, Hermes (traite IC2), 2004. http://www.lavoisier.fr/livre/notice.asp?ouvrage=2138008

J.-M. Muller. *Arithmétique des Ordinateurs*, Masson, 1989. Épuisé, disponible sur http://prunel.ccsd.cnrs.fr/ensl-00086707

H. S. Warren. *Hacker's Delight*, Addison-Wesley, 2002.

Structures de données

R. D. Necaise. *Data Structures and Algorithms Using Python*, John Wiley & Sons Ltd, 2011.

Algorithmique et complexité

T. Cormen, C. Leiserson, R. Rivest. *Introduction à l'algorithmique*, Dunod, 1994.

M. L. Heltand. *Python Algorithms : Mastering Basic Algorithms in the Python Language*, APress, 2010.

D. E. Knuth. *The Art of Computer Programming. Volume 1 : Fundamental Algorithms*, third edition, Addison-Wesley, 1997. *Volume 2 : Seminumerical Algorithms*, third edition,

Addison-Wesley, 1997. *Volume 3 : Sorting and Searching*, second edition, Addison-Wesley, 1998. *Volume 4A : Combinatorial Algorithms, Part 1*, Addison-Wesley, 2011.

R. Sedgewick et K. Wayne. *Algorithms*, fourth edition, Addison-Wesley, 2011.

Ingénierie numérique

G. Debeaumarché, J. Moisan. *Mathématiques et informatique*, Ellipses, 1988.

N. J. Higham. *Accuracy and stability of numerical algorithms*, Siam, 1995.

M. Schatzman. *Analyse numérique - une approche mathématique*, Dunod, 2001.

Pivot de Gauss

W. Appel. *Probabilités pour les non probabilistes*, H&K, 2013.

S. Chang, G. Jost, S. Yoon. *Parallelization of Gauss-Seidel Relaxation for Real Gas Flow*, NAS technical report, juin 2005.

F. Chatelin. *Valeurs propres de matrices*, Masson, 1988.

P. G. Ciarlet. *Introduction à l'analyse matricielle et à l'optimisation*, Masson, 1980.

Résolution numérique d'équations sur les réels

R. A. Holmgren. *A first course in discrete dynamical systems*, Springer-Verlag, 1994.

V. Pan, R. Schreiber. *An Improved Newton Iteration for the Generalized Inverse of a Matrix, with Applications*, SIAM J. Sci. and Stat. Comput., 12(5), pp. 1109–1130.

Résolution numérique d'équations différentielles

V. Arnold. *Équations différentielles ordinaires (et compléments)*, MIR - Librairie du globe, 1996.

M. Hirsch, S. Smale. *Differential equations, dynamical systems, ans linear algebra*, Academic Press, 1974.

SCOPOS. *Mathématiques en situation*, Springer, 2000.

F. Verhulst. *Non linear differential equations and dynamical systems*, Springer, 1996.

Bases de données

D. Maier. *The Theory of Relational Databases*, Computer Science Press, 1983. Épuisé, disponible sur `http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html`

Index

affectation simultanée, 75
agrégation, 285
algèbre relationnelle, 266
algorithme, 84
alias, 152, 167
appel d'une fonction, 115
argument
  effectif, 115, 123
  fonction en tant qu', 124
  formel, 115
array, 167
arrondi
  erreur d', 49
  sur les décimaux, 48
AS, 270
assert, 126, 202
attribut, 258

base
  2, 37
  16, 36
  numération, 34
base de données
  architecture, 271
  clés, 368
  gestionnaire, 268
  interface graphique, 363
  types, 364
bibliothèque, 127
binary search, 156
bit, 8, 34
bitmap, 359
bloc, 87, 115
booléen, 34
boucle, 96
  correction, 101
  imbrication, 109, 145
  interrompre, 108
  invariant, 101
  terminaison, 98
  variant, 98
break, 108
Brent
  méthode de, 211
bus, 7

calculatrice, 352
carte mère, 338
cas de base, 132
chaîne de caractères, 77, 152
  str, 79
clé, 276
  auto-incrémentée, 371
  étrangère, 277
  primaire, 276, 368
close, 376
commentaires, 90
compilateur, 24
complexité, 146, 187, 203
  amortie, 149, 226

au pire, 148
dans le meilleur des cas, 149
en espace, 149
en moyenne, 149
en temps, 146
optimale, 326
compréhension
tableau par, 151, 231
conditionnement, 191
corps d'une fonction, 115
correction, 202
d'une boucle, 101
d'une fonction récursive, 135

débogueur, 30, 103, 118, 137
décomposition
LU, 174, 182, 195
QR, 196
de Bruhat, 174, 182, 195
de Choleski, 196
définition d'une fonction, 114
dérivée (évaluation), 207, 212
dichotomie, 200
diviser pour régner, 156
diviseurs d'un entier, 144
division cartésienne, 279
domaine, 258
droits d'accès, 20

échange
fichier d', 335
`elif`, 92
embarqué
système, 7
enregistrement, 258
ensemblistes
opérateurs, 261
`enumerate`, 154
entrées/sorties
`input`, 72
`print`, 85
Euclide
algorithme d', 354
Euler
méthode d'
implicite, 241
méthode d', 220

`EXCEPT`, 270
exponentiation rapide, 104, 134

factorielle, 104
Fermat
méthode de, 356
fichier
lecture, 375
système de, 16
écriture, 377
file, 300
firmware, 12
flocon, 142
fonction, 114
anonyme, 125, 203, 227
appel de, 115
arguments d'une, 115
comme argument, 124
d'agrégation, 286
locale, 124
mutuellement récursives, 135
partielle, 126
récursive, 130
totale, 126
`for`, 104
`fractions`, 194, 195

Gauss
méthode de Gauss-Seidel, 198
pivot de, 173
Gilbreath, 314
Givens
méthode de, 196
`GROUP BY`, 292

Halley
méthode de, 210
`HAVING`, 292
`help`, 117
Heun
méthode de, 223, 225, 239, 242
Horner
méthode de, 154
Householder
méthode de, 196

IEEE 754 (norme), 45

if, 88
elif, 90
else, 90
imbriqués, 92
Image, 194, 379
image
conversion, 360
couleurs, 362
lecture, 379
traitement, 360
écriture, 380
import, 127
in, 107
indentation, 88, 97, 115
inode, 19
input, 72
instruction, 9, 85
boucle conditionnelle, 96
boucle inconditionnelle, 104
conditionnelle (test), 88
de branchement, 9
for, 104
if, 88
séquence, 86
while, 96
interpréteur, 24
INTERSECT, 270
invariant, 194, 202
itérable, 106

Jacobi
méthode de, 197
JOIN, 290
join, 129
jointure symétrique, 281

Knuth
mélange de, 168

labyrinthe parfait, 310
lambda, 125, 203, 226
len, 129
linalg, 186
choleski, 196
eigvals, 193
inv, 194
jacobi, 197
lu, 195
solve, 186, 188
liste, 150, 304

map, 168
matplotlib.pyplot, 232
clf, 234
plot, 232
savefig, 232
show, 232
matrice, 160
copie, 163
création, 162
de Hilbert, 186, 193, 194
de Virginie, 190, 193, 194
dimensions, 164
produit de, 165
transposition, 164
mélange, 314
de Knuth, 168
mémoire
de masse, 12
morte, 12
virtuelle, 334
vive, 7
métadonnée, 19
méthode, 129
microcontrôleur, 7
module, 127
modèle relationnel, 257
mot mémoire, 8, 34
multitâche, 14

Newton
méthode de, 200, 204
nombres à virgule flottante
dépassement de capacité, 47
erreur d'arrondi, 48
exposant, 45
mantisse, 45, 191
test à zéro, 49
None, 116, 155
notation polonaise inverse, 308
NPI, 308
numpy, 185
arange, 231
array, 167, 229

linspace, 231
roots, 211
vectorize, 232
n-uplet, 74

$O(\ldots)$, 146
objet, 129
langage orienté, 129
octet, 8, 37
open, 376
ordre d'évaluation, 121

parenthèses
langage de, 306
pas
choix du, 224
pas (choix du), 225
passage
d'arguments, 116
par valeur, 122
pendule
amorti, 246
non amorti, 236
périphériques, 8
permission, 20
phpMyAdmin, 363
pile, 299
sommet, 300
pipeline, 10, 198
pivot, 176
de Gauss, 173
partiel, 179
portée, 120
portrait de phase, 235, 242, 246
précondition, 126
print, 85
processeur, 8
produit cartésien, 279
programme, 84
exécuter un, 333
projection, 263
pseudo-aléatoire, 95, 119

RAM, 8
random, 95
range, 107
rank, 179

readline, 376
recherche
d'un mot dans un texte, 158
dichotomique
dans un tableau, 156
séquentielle, 155
récurrence
démonstration par, 132, 135
forte, 134, 136
récursivité, 130
registre, 9
regroupement, 287
relation, 258
clé, 276
relationnel(le)
algèbre, 266
calcul, 261
modèle, 257
schéma, 258
renommage, 266
représentation des nombres
binaire, 37
complément à deux, 39
entiers longs, 42
virgule flottante, 45
requête, 260
croisée, 283
return, 116
reversed, 154
ROM, 12
RSA, 358
Runge-Kutta
méthode de, 223, 225, 239

schéma relationnel, 258
compatible, 262
scipy.integrate
odeint, 231
scipy.optimize
bisect, 211
brentq, 211
fsolve, 211
newton, 211
sécante
méthode de la, 214
SELECT, 269
sélection, 264

séquence d'instructions, 86
shell, 15
sommet d'une pile, 300
`split`, 130, 376
SQL, 269, 373
`str`, 79
`strip`, 377
structure de données, 299
swap, 335
système d'exploitation, 13
système embarqué, 7

table, 258, 363
tableau, 150, 256
    copie de, 152
    par compréhension, 151
    parcours, 153
terminaison, 202, 207
    d'une boucle, 98
    d'une fonction récursive, 135
terminal, 15
tortue, 380
tours de Hanoï, 141
tri, 315
    complexité optimale, 316
    fusion, 322
    par insertion, 316
    rapide, 318
`tuple`, 74, 81
Turing
    machine de, 6

`turtle`, 380
type, 56
    `bool`, 64
    `float`, 60
    `int`, 57
    `list`, 79
    `str`, 79
    `tuple`, 74

`UNION`, 270
unité arithmétique et logique, 9

valeur
    de première classe, 124
    immuable, 163
    itérable, 106
variable
    affectation, 71
    déclaration, 70
    échanger, 75
    globale, 119
    initialisation, 70
    locale, 119
variant, 98
von Neumann
    architecture de, 7

`WHERE`, 269
`while`, 96

zéro
    comparaison à, 174, 178, 195, 352

Docteur en informatique, **Benjamin Wack** est professeur agrégé de mathématiques à l'université Joseph Fourier de Grenoble. Enseignant-chercheur au LRI (université Paris-Sud), **Sylvain Conchon** enseigne la programmation et la compilation à l'université Paris-Sud. Docteur en informatique, **Judicaël Courant** enseigne les mathématiques en MPSI au lycée La Martinière-Monplaisir à Lyon. Docteur en informatique, **Marc de Falco** est professeur en MPSI au lycée international de Valbonne dans une classe pilote en partenariat avec Inria. Chercheur Inria, **Gilles Dowek** a longtemps enseigné l'algorithmique et la programmation à l'École Polytechnique ; ses travaux portent sur les liens entre le calcul et le raisonnement. Chercheur au CNRS en poste au LRI (université Paris-Sud), **Jean-Christophe Filliâtre** enseigne l'informatique à l'École Polytechnique et à l'École normale supérieure. **Stéphane Gonnord** enseigne en classe de MP au lycée du Parc à Lyon.

L'enseignement de l'informatique est indispensable à la formation scientifique de l'étudiant qui se destine à une carrière d'ingénieur ou de chercheur. Comprendre le fonctionnement des systèmes qui nous entourent et, mieux encore, savoir les programmer, permettra aux jeunes citoyens d'être acteurs du monde contemporain.

Après une introduction à l'architecture d'un ordinateur, on présente les notions clés de l'algorithmique en s'attachant systématiquement à démontrer la correction des algorithmes et à évaluer leur complexité. On étudie ensuite la traduction d'algorithmes numériques abordés en cours de mathématiques vers un langage de programmation (Python), les limitations introduites par le passage sur machine et l'utilisation raisonnée de bibliothèques de calcul. On s'initie également aux bases de données, représentation de l'information plus complexe et présente dans des applications industrielles. Enfin on aborde des concepts plus élaborés tels que la récursivité, la structure de pile et les algorithmes de tri pour donner une vision large de l'algorithmique et de la programmation.

Ce cours comporte des sections de savoir-faire qui permettent d'acquérir les capacités essentielles, des exercices de difficultés échelonnées, avec corrigé lorsque nécessaire, ainsi que des sujets de travaux pratiques. L'ensemble du manuel a vocation à être réutilisé pour le développement des travaux d'initiative personnelle encadrés (TIPE).

À qui s'adresse cet ouvrage ?

Ce manuel de cours est destiné aux élèves de première et deuxième années de classes préparatoires aux grandes écoles scientifiques, et à leurs enseignants, voies MP, PC, PSI, PT, TPC, TSI, hors BCPST, TB et ATS.

Au sommaire

ARCHITECTURE LOGICIELLE ET MATERIELLE • Machine, système d'exploitation, environnement de développement • Qu'est-ce qu'un ordinateur ? • Notion de système d'exploitation • Environnement de développement intégré • Représentation des nombres • Entiers naturels • Entiers relatifs • Nombres à virgule (flottants) • **ALGORITHMIQUE ET PROGRAMMATION** • Expressions : types et opérations • Expressions, types simples • Variables • Types composés • Instructions ; langage minimal de l'algorithmique • Instructions • Instructions conditionnelles • Boucles conditionnelles • Boucles inconditionnelles • Fonctions • La notion de fonction • Mécanismes avancés • La récursivité • Notions de complexité ; algorithmique élémentaire sur les tableaux • Complexité d'un algorithme • Structure de tableau • Recherche dans un tableau • Recherche d'un mot dans un texte • Matrices • Mode de passage des tableaux • **INGENIERIE NUMERIQUE ET SIMULATION** • Pivot de Gauss et résolution de systèmes • Principe et mise en œuvre • Complexité • Conditionnement d'une matrice • Résolution numérique d'équations sur les réels • Recherche dichotomique • Méthode de Newton • Quelle méthode choisir ? • Résolution numérique d'équations différentielles • La méthode d'Euler • Mise en œuvre • Les bibliothèques scipy et matplotlib • **BASES DE DONNEES** • Algèbre relationnelle • Limites des structures de données plates • Représentation dans le modèle relationnel • Opérateurs • Gestionnaire de base de données relationnelle • Architecture logicielle • Base de données relationnelle • Clé primaire • Opérateurs complexes de l'algèbre relationnelle • Traduction en langage SQL • **ALGORITHMIQUE ET PROGRAMMATION AVANCEES** • Structure de pile • Opérations caractéristiques • Réalisation • Applications • Algorithmes de tri • Tri par insertion • Tri rapide • Tri fusion • **TRAVAUX PRATIQUES** • Création de programmes autonomes • Mémoire virtuelle • Démontage d'un PC • Résolution d'une équation du second degré avec gestion de la comparaison à zéro • Représentation des nombres dans les calculatrices scientifiques • Arithmétique et cryptographie • Manipulation d'images bitmap • Prise en main de phpMyAdmin • Clés primaires et clés étrangères • **COMPLEMENTS PRATIQUES** • Lecture et écriture dans des fichiers • Lecture et écriture dans des images • Utilisation du module graphique turtle.



32 €



Matériel supplémentaire sur www.informatique-en-prepas.fr

www.editions-eyrolles.com
Groupe Eyrolles | Diffusion Geodif